UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Al Loro:
Lector de *feeds RSS* para asistente de voz

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Alejandro Gómez Noé

Tutor: Vicente Pelechano Ferragud

Curso 2020-2021

Resum

Últimament, els assistents de veu estan tornant-se molt populars (*Amazon Alexa*, *Assistent de Google*, etc.), i ofereixen als desenvolupadors ferramentes per a crear les seues pròpies aplicacions compatibles amb aquests sistemes. Concretament, en Alexa es diuen *"skills"*, i és possible instal·lar-les solament amb la veu. Un tipus de *skill* són els butlletins de notícies, els quals funcionen a través de *feeds RSS/Atom*. El principal problema amb elles és que, per a què sigan compatibles amb Alexa, aquestes *feeds* han de tindre un format molt precís, i és molt inusual que ho complisquen. Per aquest motiu, l'objectiu d' aquest treball és el d'implementar una *skill* on pugues afegir les teues *feeds* favorites, i escoltar notícies, blogs, podcasts... a través d'aquesta *skill*. A l'hora d'afegir una *feed*, podràs configurar aspectes com ara el seu nom, el seu idioma (amb soport per a espanyol i anglès), el màxim de notícies que et deuria llegir, o quant text deuria llegirte d'una vegada. La configuració es realitzarà a través d'una app, on podràs gestionar les teues *feeds*.

Paraules clau: Alexa, assistent de veu, Alexa skills, RSS, Atom, notícies, blogs, podcasts

Resumen

Últimamente, los asistentes de voz se están volviendo muy populares (*Amazon Alexa*, *Asistente de Google*, etc.), y ofrecen a los desarrolladores herramientas para crear sus propias aplicaciones compatibles con estos sistemas. Concretamente, en Alexa se llaman *"skills"*, y es posible instalarlas sólo con la voz. Un tipo de *skill* son los boletines de noticias, los cuales funcionan a través de *feeds RSS/Atom*. El principal problema con ellas es que, para que sean compatibles con Alexa, estas *feeds* deben tener un formato muy preciso, y es muy inusual que lo cumplan. Por este motivo, el objetivo de este trabajo es el de implementar una *skill* donde puedas añadir tus *feeds* favoritas, y escuchar noticias, blogs, podcasts... a través de dicha *skill*. A la hora de añadir una *feed*, podrás configurar aspectos como su nombre, su idioma (con soporte para español e inglés), el máximo de noticias que te debería leer, o cuánto texto debería leerte de una vez. La configuración se realizará a través de una app, donde podrás gestionar tus *feeds*.

Palabras clave: Alexa, asistente de voz, Alexa skills, RSS, Atom, noticias, blogs, podcasts

Abstract

Lately, voice assistants are becoming really popular (*Amazon Alexa*, *Google Assistant*, etc.), and they offer developers tools to make their own applications compatible with these systems. More specifically, in Alexa they are called *"skills"*, and it is possible to install them just by voice. One kind of *skill* are news briefers, which work through *RSS/Atom feeds*. The main issue with them is that, for them to be compatible with Alexa, these *feeds* must have a very precise format, and it is very unlikely to find any that complies with it. For this reason, this project's objective is to implement a *skill* where you can add your favorite *feeds*, and listen to news articles, blogs, podcasts... through said *skill*. When adding a *feed*, you will be able to configure aspects such as its name, language (with support for english and spanish), the maximum number of news articles it should read to you, or how much text it should read at once. This configuration will be done through an app, where you will be able to manage your *feeds*.

Key words: Alexa, voice assistant, Alexa skills, RSS, Atom, news, blogs, podcasts

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Durante los últimos años, Internet ha ido formando cada vez más parte de nuestras vidas, especialmente a la hora de obtener información (noticias, entretenimiento, etc.). Medios más tradicionales como la televisión o la radio han ido perdiendo relevancia frente a las redes sociales, y estos se han visto obligados a adaptarse a las nuevas tecnologías.

Por otro lado, los dispositivos móviles (como teléfonos o tabletas inteligentes) se han convertido en nuestro principal medio de consumo de contenido a través de Internet, debido a su facilidad de uso y portabilidad. Esto podemos observarlo gracias a la popularidad de las *apps*, y por tanto, del desarrollo móvil.

Pero, desde hace bastante poco, y gracias a la mejora exponencial de los sistemas de procesamiento del lenguaje natural, un nuevo competidor ha llegado al mercado: los asistentes de voz, que permiten hacer todo tipo de cosas sin mover un dedo. Empresas del calibre de Amazon y Google están invirtiendo muchos recursos en este campo con sus respectivas soluciones: Amazon Alexa y el Asistente de Google.

Ambas están disponibles para dispositivos móviles, pero también existen unos dispositivos diseñados específicamente para estos asistentes de voz: Amazon Echo y Google Nest.

Para el presente trabajo, nos centraremos en los servicios de Amazon: el asistente Alexa y los dispositivos Echo (podemos observar uno de sus modelos en la figura 1.1).

Figura 1.1: Dispositivo Amazon Echo Dot (4ª Generación, con reloj)

Gracias a estos asistentes, podemos hacer cosas como: crear alarmas, recordatorios; preguntar acerca del tiempo, el tráfico, o las noticias del día... entre otras.

Además, los asistentes de voz tienen una especial integración con la domótica: bombillas, enchufes, termostatos... son algunos ejemplos de dispositivos en los que algunas empresas han incluido soporte para poder ser controlados desde Alexa.

Pero la cosa no acaba ahí: también permite a los desarrolladores crear sus propias aplicaciones compatibles con Alexa, llamadas *skills*.

1.1 Motivación

Hace unos meses adquirí un dispositivo Echo Dot, y empecé a experimentar con sus diferentes funcionalidades. Como estudiante de Ingeniería de Software, me llamó especialmente la atención el desarrollo de skills y las posibilidades que ello permite.

Tras investigar un poco al respecto, descubrí que las skills funcionan mediante el servicio *AWS Lambda*. En su página principal en español[1] lo definen como «un servicio informático sin servidor que le permite ejecutar código sin aprovisionar ni administrar servidores». Este tipo de servicios son conocidos como *serverless*, y se han vuelto bastante populares en los últimos años.

Los servicios en la nube son muy demandados, útiles e interesantes; y considero una buena inversión de cara al futuro poder aprender más acerca de ellos. Además, este proyecto sirve como introducción en un campo novedoso y con (probablemente) mucho futuro por delante como es el de los asistentes de voz.

Por otro lado, el desarrollo de la skill me ha servido para aprender sobre un lenguaje de programación del que he oído hablar mucho últimamente: *TypeScript*[2]. Se trata de una extensión del lenguaje *JavaScript*, al que le añade un sistema de tipos. Esto, junto a otras características del lenguaje, mejora sustancialmente la experiencia de desarrollo, y ha llevado a muchas grandes empresas a migrar su código *JavaScript* hacia *TypeScript* (además de haber propiciado una mayor adopción por parte del público general).

1.2 Objetivos

El principal objetivo de este trabajo es **documentar el proceso de desarrollo de una skill**, poniendo énfasis en el uso de técnicas y herramientas ya establecidas en el desarrollo de otro tipo de aplicaciones más comunes.

Uno de los mayores impedimentos a la hora de introducirse en el entorno de Alexa es el trabajo previo de investigación que requiere. Por ello, se busca **proporcionar una guía que sirva de referencia para implementar skills**, y que se centre en los siguientes aspectos:

- Sus similitudes con el desarrollo *backend*, mediante el uso de los servicios de *AWS* y el modelo *serverless*.
- Las opciones a la hora de integrar o complementar una skill con otro tipo de aplicaciones y servicios (*apps*, bases de datos, etc.)
- El funcionamiento (a nivel del desarrollador de skills) de una interfaz guiada por voz, y cómo diseñar un modelo conversacional.

Uno de los aspectos más útiles en una guía o documentación es la existencia de **ejemplos prácticos** extraídos de un **producto software funcional**, que permitan al desarrollador obtener una visión más completa del propósito de cada pequeña funcionalidad que se trata en la guía.

Para ello, se plantea el objetivo de **implementar una skill** que **solucione un problema real** o **mejore un modelo existente**. En el contexto de este trabajo se tratará de «*Al Loro*», una skill de noticias que proporcionará una mejor integración con las feeds *RSS* con respecto de la funcionalidad existente en el propio sistema de Alexa.

[1] https://aws.amazon.com/es/lambda/
[2] https://www.typescriptlang.org

1.3 Estructura

Los próximos capítulos seguirán la siguiente estructura:

- **Estado del arte**

  Se hará una comparativa de los principales asistentes de voz disponibles en el mercado, comentando sus fortalezas y debilidades, y se justificará la elección de asistente para este trabajo.

- **Análisis del problema**

  Se comentará la dificultad para desarrolladores que la documentación en este trabajo del desarrollo de "Al Loro" puede ayudar a aliviar, y el problema con la lectura de noticias en Alexa del que nació la skill como posible solución.

- **Diseño de la solución**

  Se repasará la arquitectura del sistema que conforma "Al Loro", profundizando en los diferentes subsistemas y el proceso de decisión de las tecnologías utilizadas en los mismos.

  También se analizará con mayor detalle el diseño de la skill en sí misma: la estructura del proyecto, las diferentes partes por las que está formado (interfaz, lógica, persistencia) y cómo se relacionan estos conceptos en el desarrollo de skills.

- **Desarrollo de la solución propuesta**

  Se revisará el proceso de desarrollo del sistema, analizando las diferentes librerías y herramientas utilizadas, así como el proceso de aprendizaje y los contratiempos que surgieron durante la implementación.

- **Implantación**

  Se explicará el funcionamiento de la fase de certificación y publicación de una skill en Alexa. Además, se mostrará un proceso similar en el contexto de la publicación de la app complementaria en la *Google Play Store* (la tienda de aplicaciones para Android).

- **Pruebas**

  Se introducirán los principales conceptos relacionados con la fase de pruebas del proyecto (las pruebas unitarias y el análisis estático), junto con las herramientas utilizadas para ponerlos en práctica.

- **Conclusiones**

  Se resumirá el contenido de los capítulos anteriores, relacionándolo con los objetivos iniciales y con asignaturas cursadas en el grado.

Para comprender mejor aquellas palabras o expresiones que sean técnicas o específicas de la temática de este trabajo y se repitan a lo largo del mismo, existe un **glosario** al final del documento.

CAPÍTULO 2
Estado del arte

2.1 Asistentes de voz

Existen tres principales asistentes de voz en el mercado: Asistente de Google[1], Amazon Alexa[2], y Siri[3] (de la compañía Apple). La tabla 2.1 muestra una comparativa entre ellos.

	Google	**Alexa**	Siri
Kits de desarrollo	*Actions* [1]	*Skills* [2]	*SiriKit* [3]
Lenguajes soportados	JavaScript (Node.js)	JavaScript (Node.js), Python	Swift, Objective-C
Plataformas	Android, Google Home	Amazon Echo, Fire TV, Android e iOS[4]	iOS, HomePod
Mayor enfoque	Búsqueda por Internet	*Skills*, domótica, entretenimiento	Productividad

Tabla 2.1: Comparativa de los asistentes de voz más populares

2.1.1. Asistente de Google

Se trata de uno de los asistentes de voz más populares en el mercado, puesto que se encuentra instalado por defecto en el sistema operativo con mayor presencia en dispositivos móviles: Android.

Su principal atractivo es su sistema de **búsqueda por Internet**. Google cuenta con el motor de búsqueda más utilizado, y lo aprovecha al máximo integrándolo en el asistente.

Su kit de desarrollo, ***Google Actions***, busca facilitar el aprendizaje y mejorar la experiencia de desarrolladores **principiantes** y entusiastas. Integrando en su consola una herramienta de modelado basada en **diagramas** (figura 2.1), el proceso de desarrollo obtiene un enfoque más **visual**, lo cual ayuda a introducirse en el flujo de trabajo habitual de este tipo de programas: uno más centrado en el **modelo conversacional** y el **árbol de decisiones**.

[1] https://assistant.google.com/intl/es_es/
[2] https://developer.amazon.com/es-ES/alexa
[3] https://www.apple.com/es/siri/
[4] Es necesario instalar la app de Amazon Alexa para poder utilizar dicho asistente en Android e iOS.

Aun así, no deja de lado a los desarrolladores más avanzados, ya que provee de una funcionalidad llamada «***webhooks***» con la que poder integrar una API personalizada, ya sea mediante un *endpoint* HTTPS o haciendo uso del servicio «*Cloud Functions*» que ofrece la plataforma Firebase (un servicio de Google).

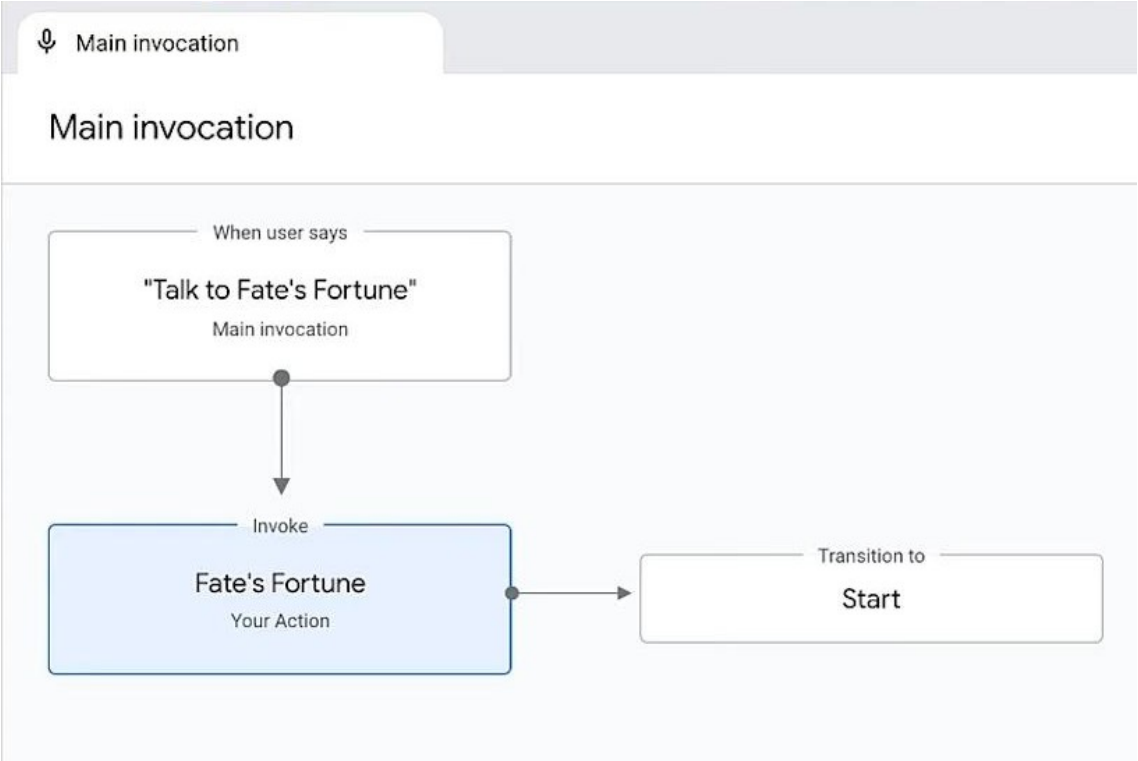


Figura 2.1: Herramienta de modelado basada en diagramas - *Google Actions Console*

2.1.2. Amazon Alexa

Pese a tener el menor uso en dispositivos móviles de los tres, Alexa es un gran contendiente en el marco de los dispositivos *Smart Home*, gracias a su familia de productos Amazon Echo.

Su sistema de búsqueda por Internet es el peor de los tres con diferencia[5], pero trata de compensarlo con su enfoque en la **integración con otras plataformas**. Con una gran inversión en su sistema de ***skills***, Alexa busca que desarrolladores de todo tipo implementen aplicaciones para poder hacer uso de sus sistemas a través de este asistente de voz.

Entre las categorías de skills más destacadas se encuentran:

- **Entretenimiento**: Juegos y curiosidades, cuentos para niños, meditación, recetas de cocina...
- **Hogar digital** (*Smart Home*): Skills de domótica asociadas a dispositivos como bombillas inteligentes, cámaras de vigilancia, o termostatos.
- **Multimedia**: Dan acceso a servicios de música o vídeo, como cadenas de radio o *podcasts*.

[5]En un estudio [4] de la empresa *SEMrush* realizado en el año 2020, el porcentaje de preguntas que quedaron sin respuesta usando Alexa fue de un 23 %, mientras que en el resto de asistentes era menor del 10 %.

- **Noticias**: Servicios de noticias tanto de televisión, radio, periódicos, revistas, e incluso independientes (por ejemplo, a través de YouTube).

En cuanto a la plataforma de *skills*, su curva de **aprendizaje** es un **término medio** entre las otras dos. Como ofrece una integración con los lenguajes ***JavaScript*** y ***Python*** en forma de librerías especializadas, el **desarrollador web** es el más beneficiado, con una mayor facilidad de adaptación debido al uso extensivo de tecnologías web conocidas en el kit de desarrollo.

2.1.3. Siri

Este asistente dispone de un público más selecto, pues sólo se encuentra disponible en productos de la marca Apple (como el *iPhone* o el *iPad*), los cuales tienen una base de usuarios más reducida. Sin embargo, estos son más proclives a utilizar Siri, llevando a algunas encuestas a mostrar alrededor de un 45 % de cuota relativa de mercado en *smartphones* de los Estados Unidos [5].

Respecto a su conjunto de funcionalidades, no parece haber una que destaque especialmente sobre el resto en cuanto a su promoción. Aun así, muchas comparten una misma temática: la **productividad**. Su publicidad está enfocada en la comodidad y rapidez que aporta utilizar Siri para realizar tareas cotidianas. Un ejemplo de ello son los **atajos**: un sistema para, según la página principal de Siri, «agilizar las tareas que haces más a menudo» mediante «una serie de funciones rápidas muy útiles» que encadenan una sucesión de acciones.

El kit de desarrollo de Siri (***SiriKit***) está fuertemente acoplado al ecosistema de *iOS* y su desarrollo de apps – de hecho, toda interacción con los sistemas de Siri debe ser hecha desde una app desarrollada para el sistema operativo móvil de Apple[6].

Por este motivo, esta plataforma de desarrollo es la **menos abierta** a nuevos desarrolladores; ya que, como mínimo, se requiere de:

- Un conocimiento general del lenguaje de programación (*Swift*), el cual es utilizado casi exclusivamente en productos de Apple.
- Un ordenador de la familia de Mac, para tener acceso al editor de código *XCode* y sus herramientas, sin las cuales no se pueden compilar apps[7] para iOS.

2.2 Propuesta

Este trabajo se centrará en el desarrollo de skills para *Amazon Alexa*. Como dispone de un mayor nivel de inversión y popularidad que los kits de desarrollo de la competencia, es más probable que tenga éxito de cara al futuro; lo cual hace que sea un tema de estudio interesante.

Al tratarse de un campo novedoso, es muy probable que muchos desarrolladores con otras especialidades (desarrollo web o móvil, por ejemplo) empiecen a interesarse por Alexa y traten de adaptar sus sistemas existentes a este nuevo medio. Por este motivo, en los próximos capítulos se va a aportar una nueva visión desde esta perspectiva.

[6] También existen las «*app extensions*» (https://developer.apple.com/documentation/sirikit/intent_handling_infrastructure/structuring_your_code_to_support_app_extensions), intermediarios que actúan en nombre de la app; pero suelen requerir de un alto grado de integración con la propia app de iOS.

[7] Algunos frameworks multiplataforma como *Flutter* permiten crear apps para Android e iOS, pero sin *XCode* no se puede generar el producto final en iOS.

CAPÍTULO 3
Análisis del problema

3.1 Lectura de noticias

Entre los modelos de skill pre-construidos que Alexa proporciona (figura 5.3), se encuentra el modelo de noticias [10], el cual no requiere ningún conocimiento de programación, sino un simple enlace a una *feed* con el estándar *RSS* o con formato *JSON* y una estructura concreta. Técnicamente, este modelo permite añadir contenido de blogs, páginas de noticias, etc. para que Alexa lo lea. Pero también tiene varios problemas que presentamos en las siguientes subsecciones:

3.1.1. Intención de uso del modelo de noticias

Este sistema no está pensado para que un usuario añada el contenido de sus webs favoritas, sino para que los dueños de estas webs creen sus propias skills con este contenido para que luego sus usuarios las puedan instalar y consumirlo.

De hecho, no se puede publicar una skill de este tipo salvo que quien lo haga sea dueño del contenido que incluye, o disponga de permiso para distribuirlo de esta forma [11].

En principio, un usuario podría crear estas skills y mantenerlas para uso personal (pues sólo estarían disponibles al público tras un riguroso proceso de certificación [12]), pero sería poco conveniente, pues tendría que: o crear una skill para cada uno de los servicios de noticias que utilice, lo cual sería un proceso tedioso y poco razonable; o tener una sola skill para todos sus servicios, caso en el que tendría poco control sobre cuál de ellos desea escuchar en cada momento. Además, si no es un desarrollador, es poco probable que sepa de esto (ni quiera hacerlo).

3.1.2. Formato esperado de una feed

El modelo requiere una feed *RSS* (o *JSON*). En términos muy simples, se trata de una lista con los diferentes artículos, ya sean noticias, entradas de un blog, etc. Estos artículos tienen varios campos. Los más importantes serían el título y la descripción.

Muchos sitios incluyen el título de la noticia en el campo "título", y el subtítulo en el campo "descripción". Si lees una noticia, el subtítulo no es más que una extensión del título, dependiente de este.

Pues bien, el modelo de noticias de Alexa únicamente lee al usuario el campo "descripción", y no el campo "título" [13]. Esto lleva a que, cuando te lee una noticia, esta queda inconclusa, puesto que te lee el subtítulo, pero no el título.

En el caso de los blogs no es tan grave, ya que en el campo "descripción" suelen incluir el contenido de la publicación (en algunos casos completo, en otros sólo un segmento).

Pero, aun así, el título es lo más importante; puesto que te resume en pocas palabras su contenido, y te sirve para decidir rápidamente si te interesa o no. Es más, en algunos casos, no puedes comprender el contenido de la publicación sin leer su título –especialmente si el contenido está resumido–.

Esto lleva a situaciones en las que se le lee al usuario un contenido que resulta no interesarle, pero no lo sabe hasta que lo ha leído entero.

Por otro lado, uno de los requisitos para una feed es que su contenido se encuentre en texto plano y no contenga caracteres especiales [14]. Muchos sitios no lo cumplen, y por tanto no pueden ser utilizados para crear una skill de este tipo.

3.2 Modelado Conceptual

Los casos de uso de la skill (figura 3.1) son relativamente sencillos, pues la función principal es leer feeds, y para ello no hacen falta muchos más comandos. Obtener una lista de las feeds disponibles para saber qué feeds se pueden leer y recibir ayuda acerca de los comandos disponibles y cómo llamarlos son las otras dos funcionalidades básicas de la skill.

Sin embargo, existe una función un tanto más compleja en el contexto de una aplicación guiada por voz: gestionar las feeds. Añadir, editar o eliminar feeds no sólo es un proceso tedioso por voz, sino que no es viable porque requiere indicar enlaces web. Por tanto, se utiliza una app para proveer estos servicios. Mediante un sistema de autenticación basado en un código de seis dígitos, la app puede gestionar la lista de feeds de la que la skill hace uso.

Para la función de leer una feed, es importante mostrar un diagrama de flujo (figura 3.2) para ver con más detalle el proceso en base a la experiencia del usuario.

En otras circunstancias, el proceso podría ser más sencillo, pero debido a algunas características y limitaciones de Alexa, se han tomado varias decisiones:

- Al comenzar el proceso de lectura de una feed, el usuario puede haber proporcionado previamente su nombre al ejecutar el comando, pero también puede no haberlo hecho. Por suerte, Alexa se encarga de esta situación[1], pidiendo al usuario el nombre de la feed si no la tiene.

- Debido a una limitación en el número de caracteres permitidos en una respuesta, el contenido de feeds más largas debe ser dividido en secciones. Cada vez que acaba de leer una sección, Alexa pregunta al usuario si quiere que siga leyendo (siempre y cuando quede contenido por leer).

Entrando en un apartado más técnico, el siguiente modelo es un diagrama de clases de aquellos objetos relacionados con las feeds (figura 3.3).

La característica más llamativa de este modelo es la falta de métodos entre las clases. Esto se debe a una limitación en la forma en la que Alexa almacena los objetos entre peticiones de una misma sesión. Cada vez que una petición termina de ser manejada, aquellos objetos que quieran ser persistidos deben convertirse a texto para ser incluidos

[1] Se trata de una función que hay que activar y configurar desde la consola de Alexa, pero no hace falta gestionarlo por código.

en la respuesta; y así podrán ser recuperados en la siguiente petición. Por desgracia, durante este proceso los métodos de los objetos se pierden y dejan de ser accesibles. Es por eso que todas las funciones se definen de forma separada, y las clases sólo contienen los datos estrictamente necesarios.

Por este mismo motivo, las clases del diagrama están realmente definidas como «*interfaces*»: estructuras que indican al sistema de tipos de TypeScript cómo debería estar formado un objeto de JavaScript (el cual es dinámico y puede tener la estructura que quiera). Las interfaces son eliminadas en el proceso de compilación, y sólo sirven para mejorar la experiencia de desarrollo y detectar posibles *bugs*.

Otras particularidades del diagrama de clases son:

- La existencia de una clase «*Feed*» que deriva de otra llamada «*FeedData*» y sólo contiene un atributo de más, el nombre.

  La idea detrás de esto es que en la base de datos, el nombre está almacenado en dos atributos según el idioma del dispositivo: inglés y español.

  Esto se puede configurar desde la app, y permite al usuario cambiar el idioma de su dispositivo cuando le plazca y seguir teniendo acceso a su lista de feeds con nombres adecuados.

  Al iniciarse la skill, se detecta el idioma actual del dispositivo y se obtiene el atributo adecuado. Primero se indica al sistema de tipos que no existe aún un atributo «nombre» usando *FeedData*, y una vez añadido se redefine su tipo[2] como *Feed*.

- La relación entre las clases «*FeedData*» y «*FeedItems*». De nuevo, se debe a la limitación que no permite tener la función «*getItems(FeedData)*» como método de la clase «*FeedData*».

  En este caso, «*FeedData*» representa el conjunto de datos que el usuario ha indicado en la app (y, por tanto, se encuentra en la base de datos); mientras que «*FeedItems*» contiene el resultado de obtener las noticias de la feed, tras haber sido procesado y ordenado en un formato más cómodo para el código encargado de leer las noticias.

Por otro lado, este diagrama incluye dos funcionalidades que, aunque no son esenciales, pueden resultar bastante útiles:

- Soporte para podcasts: Algunas feeds pueden formar parte de un podcast, e incluir un enlace a un capítulo en cada elemento. En esos casos, la skill pregunta al usuario si quiere escucharlo tras haber leído todo el contenido de texto. En caso afirmativo, hace uso del sistema de reproducción de audio de Alexa para ponerlo en marcha.

- Filtros para feeds: Algunos sitios de noticias abarcan una gran variedad de temas entre su contenido, y al usuario no tienen por qué interesarle todos. Por eso, desde la app se pueden indicar una serie de textos y/o categorías para que sólo aparezcan aquellos elementos que los contengan. También se puede indicar si tienen que coincidir todos, o si cualquiera de ellos sirve para darlo por bueno.

[2] «Redefinir el tipo» sólo se refiere a cambiar la interfaz que define la estructura del objeto. El tipo «real» sigue siendo el de «objeto» a secas, así que a nivel de ejecución no tiene ningún efecto.

3.3 Solución propuesta

Visto el funcionamiento del actual modelo de noticias, es interesante plantear una posible alternativa: una skill a través de la cual el usuario pueda añadir sus propias feeds y escuchar su contenido, sin depender de que sus dueños sepan de la existencia de Alexa y decidan invertir tiempo en crear su propia skill de noticias.

Del mismo modo, se puede aprovechar su proceso de desarrollo para profundizar en los diferentes aspectos de la creación de skills en Alexa, como:

- Sus similitudes con el desarrollo *backend*, mediante el uso de los servicios de *AWS* y el modelo *serverless*.
- Las opciones a la hora de integrar o complementar una skill con otro tipo de aplicaciones y servicios (*apps*, bases de datos, etc.)
- El funcionamiento (a nivel del desarrollador de skills) de una interfaz guiada por voz, y cómo diseñar un modelo conversacional.

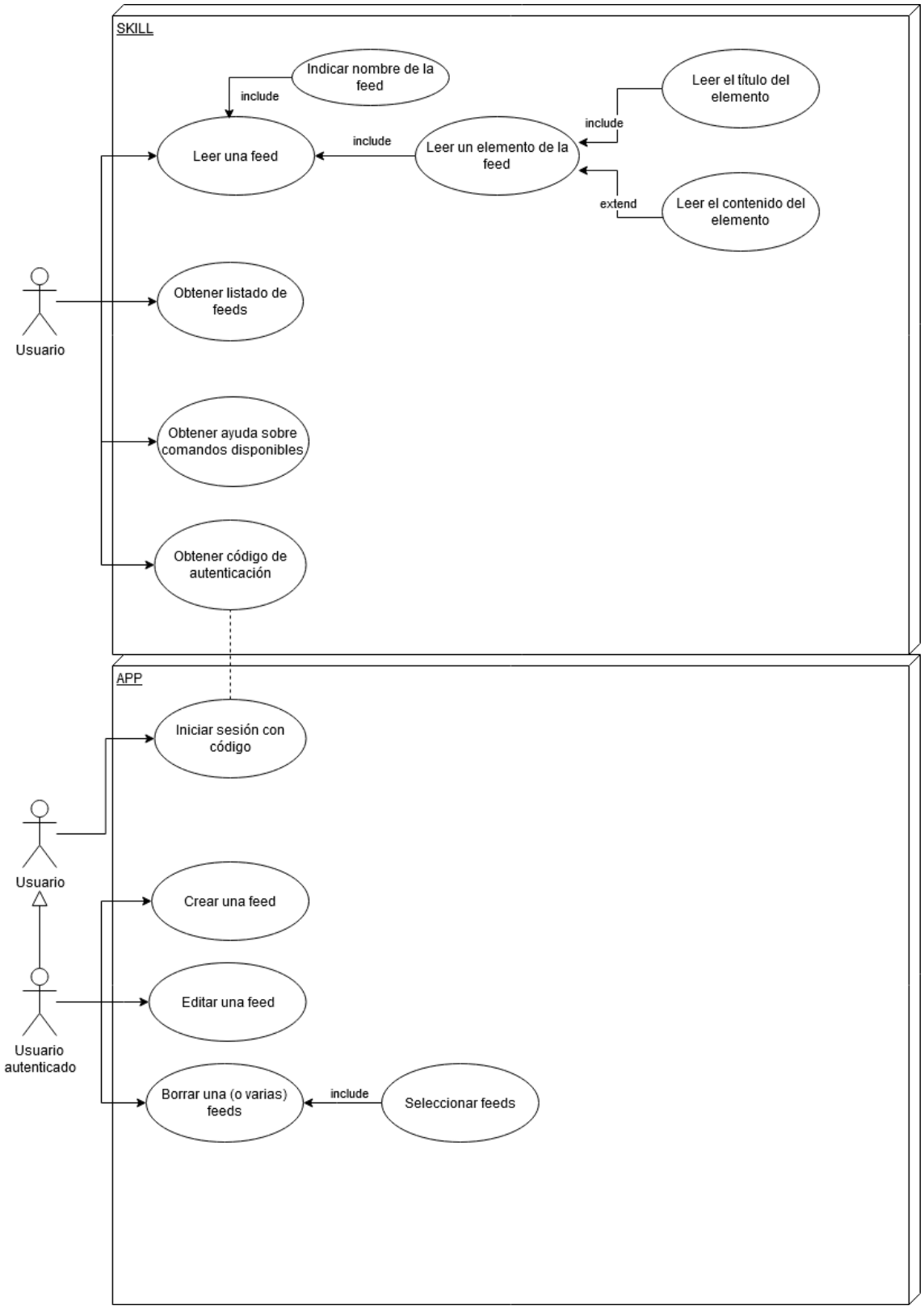


Figura 3.1: Diagrama de casos de uso

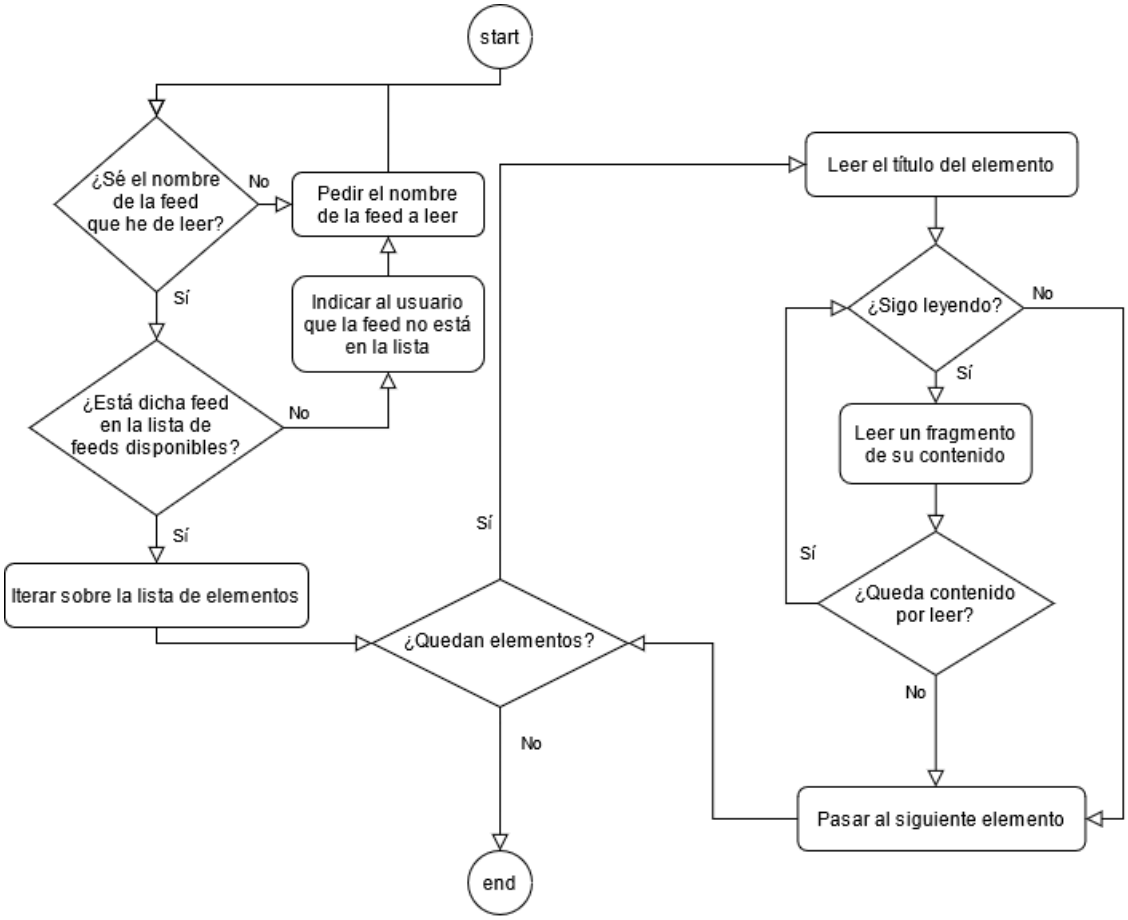


Figura 3.2: Diagrama de flujo para el caso de uso "Leer una feed"

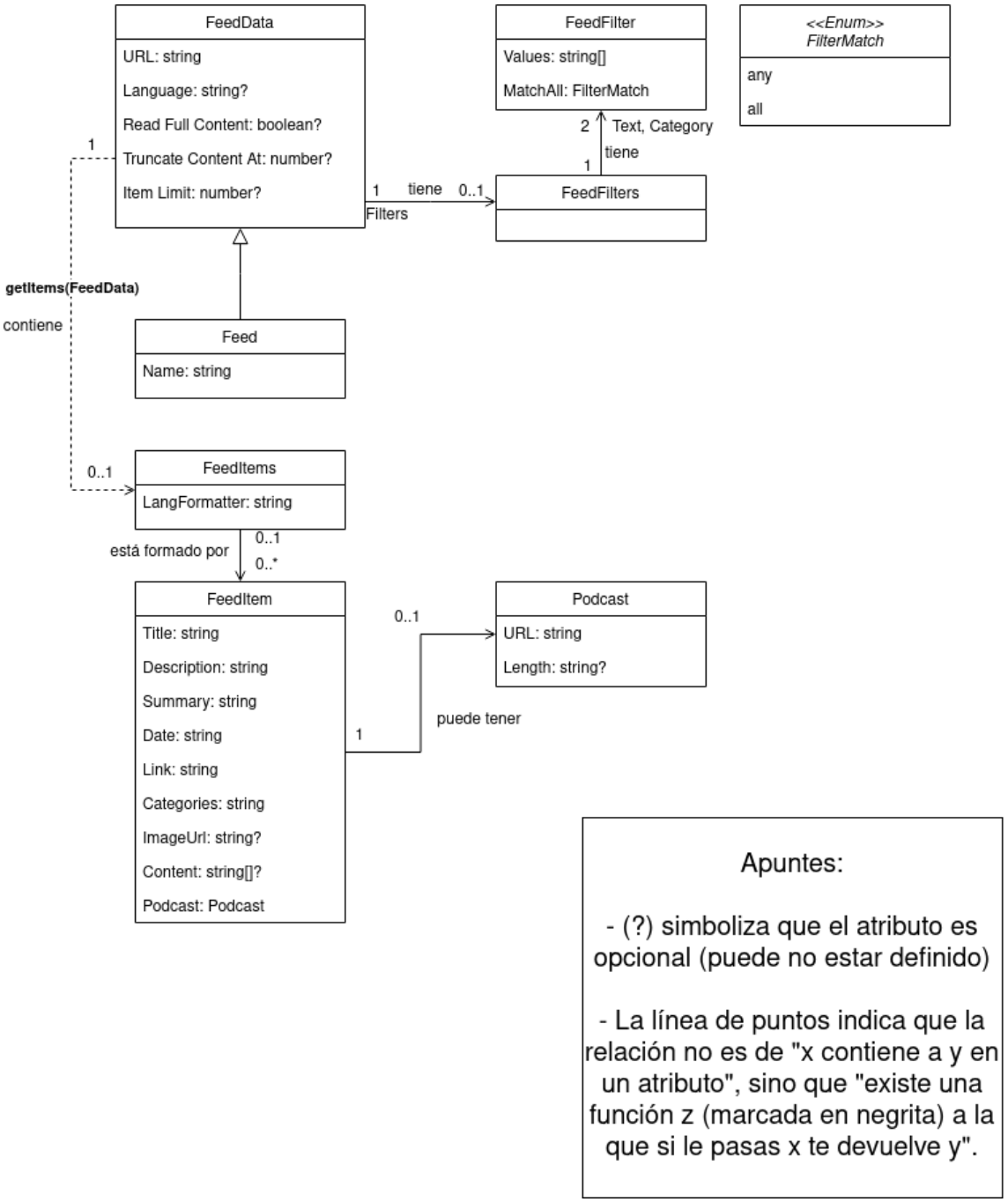


Figura 3.3: Diagrama de clases (Feed)

CAPÍTULO 4
Diseño de la solución

4.1 Arquitectura del Sistema

El sistema está dividido en tres grandes subsistemas, mostrados en la figura 4.1:

- **Amazon**: Se trata del subsistema principal, puesto que contiene al propio servicio de Alexa y la skill. Además, engloba una serie de servicios de una cuenta de AWS independiente, los cuales se encuentran integrados con la skill para añadirle la funcionalidad necesaria para comunicarse con la app.
- **Firebase**[1]: Este servicio de base de datos de Google se encarga de persistir las feeds. Además, proporciona un servicio de autenticación para poder gestionarlas a través de la app.
- **App**: Desarrollada mediante el *framework* de Google *Flutter*[2], permite al usuario gestionar sus feeds fácilmente.

4.1.1. Sistema de gestión de feeds

Otras arquitecturas contempladas

Inicialmente se buscaron opciones tratando de que toda la funcionalidad de cara al usuario pudiera existir dentro de la skill (es decir, que fuera posible interaccionar con ella mediante la voz), pero se acabó considerando que esto no era posible en el caso de la introducción de los enlaces a las diferentes feeds. Esto se debe a que el texto de un enlace no tiene por qué ser pronunciable (o fácilmente comprensible por Alexa), lo cual podría provocar frustración al usuario.

Tras verse frustrado el plan ideal, se barajaron alternativas en las que puedas introducir estos datos de manera escrita. En primer lugar, se investigó si existía alguna tecnología así entre los servicios de Alexa, para que sirviera de apoyo a la skill sin tratarse de una aplicación aparte.

Esto llevó al descubrimiento del *APL* (*Alexa Presentation Language*): un lenguaje a través del cual, según su página dedicada en la documentación de Alexa [15], puedes crear una serie de «documentos» que aportan «experiencias visuales que acompañan a la skill», con las cuales los usuarios pueden también interactuar.

[1] https://firebase.google.com/
[2] https://flutter.dev/

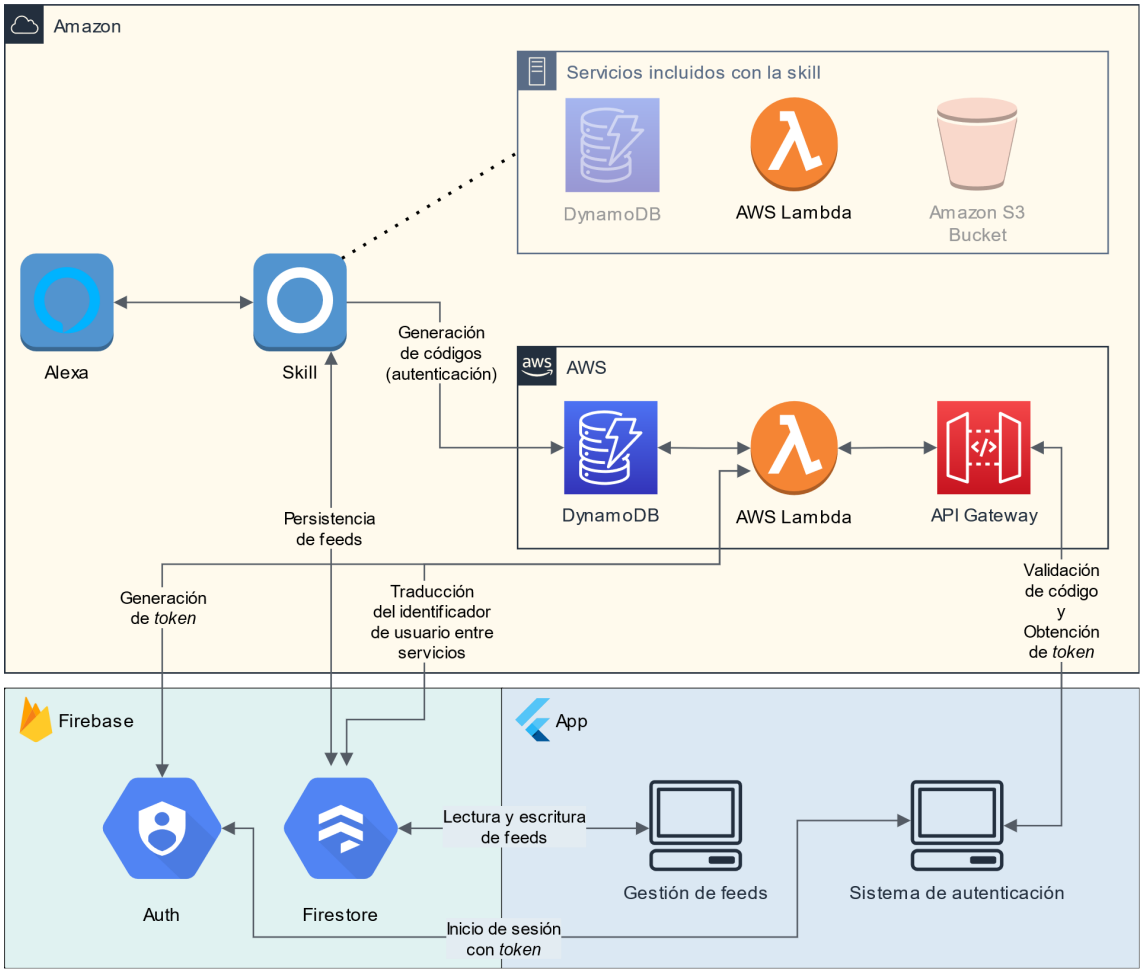


Figura 4.1: Diagrama de bloques representando la arquitectura del sistema
(los servicios en gris están disponibles pero no son utilizados)

Sin embargo, su uso se acabó tachando de inviable, debido a que está limitado a aquellos dispositivos con pantalla propia («el Echo Show, la Fire TV, algunas tabletas Fire, y otros dispositivos», según la documentación), y no parece estar disponible a través de la app Alexa[3]. Es por ello que aquella funcionalidad que aporte no debería ser esencial; es decir, todo aquello que se pueda hacer gracias a APL, debería poder hacerse también por voz.

Decisión final y sus ventajas

Tras esta investigación, se llegó a la conclusión de que la skill por sí sola no resultaría suficiente para conseguir toda la funcionalidad deseada, y sería necesaria una aplicación complementaria para la gestión de las feeds. Dicha aplicación podría ser web, de escritorio o móvil.

La aplicación de escritorio fue la primera en descartarse, pues no aporta nada que no se pueda hacer a través de una web, y estaría disponible en un menor número de dispositivos.

Entre la aplicación web y móvil, surgió la siguiente coyuntura:

[3]Se utiliza la expresión "parece" porque esto no se menciona en la documentación. Sin embargo, en el foro de Alexa existen comentarios donde se indica que no está disponible.

- Por un lado, la aplicación web sería más versátil, al estar disponible tanto para ordenadores como dispositivos móviles.
- Pero por otro lado, una aplicación móvil sería menos costosa en cuanto a horas de trabajo, al disponer de mayor experiencia en su desarrollo. Además, el uso de dispositivos móviles está muy extendido, por lo que la falta de versatilidad no sería tan grave.

Finalmente, se optó por desarrollar una aplicación móvil (o app, para abreviar).

4.1.2. Integración entre la skill y la app

Tras decidir que la existencia de una app para gestionar las feeds es necesaria, el siguiente paso es averiguar cómo integrar ambos servicios para que funcionen conjuntamente.

Base de datos

En primer lugar, se requiere un sistema de base de datos común al que puedan acceder ambas aplicaciones. Aquí se valoraron dos opciones:

- Amazon dispone del servicio *DynamoDB*, accesible por parte de la skill desde su creación. Sin embargo, la integración de éste con la app resulta bastante más compleja, y su curva de aprendizaje es relativamente alta.
- En cambio, el servicio de Google ***Cloud Firestore*** (de la familia de productos *Firebase*) tiene una integración más sencilla con la app, y acceder a él a través de la skill no tiene mucha complicación.

Autenticación

En segundo lugar, es necesario un sistema de autenticación para la app. Aquel que se utilice debe ser compatible tanto con la base de datos como con la skill, ya que esta última es la que proporciona identificadores de usuario. Además, el proceso de inicio de sesión debe ser fácil y cómodo para el usuario. Y sería preferible que no tenga que proporcionar credenciales de ninguna cuenta que pueda provocar reticencia al usuario, como Google o Amazon[4].

Como resultado, se ideó un sistema de autenticación mediante un **código** de 6 dígitos. El procedimiento es el que sigue:

1. Desde la skill, el usuario indica que desea iniciar sesión. Alexa genera y dicta el código que podrá utilizar para ello. Internamente, dicho código se ha asociado a su usuario mediante un identificador y se ha almacenado en una base de datos (*DynamoDB*).
2. El usuario accede a la app, e introduce el código que Alexa le ha dictado. Por su parte, la app realiza una petición a una *API*, que a su vez se dedicará a validar que el código se encuentra en la base de datos que la skill utiliza para guardar los códigos.

[4]Este argumento surgió a partir de un comentario presente en la página de una skill similar a esta, donde era tildada de "estafa" por requerir un inicio de sesión en Google para utilizar una página web complementaria.

En caso afirmativo, se generará un *token* que la app recibirá como respuesta, y con el que podrá iniciar sesión a través del servicio de autenticación correspondiente.

Respecto a los servicios disponibles para las bases de datos anteriormente mencionadas, las opciones son:

- Por parte de Amazon, se encuentra *Cognito*[5]. Este servicio forma parte de *AWS*. Posee una amplia variedad de funcionalidades, pero al mismo tiempo resulta más complejo.
- Entre los productos de Google, ***Firebase Authentication***[6] (o *Auth* para abreviar) sería la solución. Es más sencillo de configurar que el anterior, y posee una opción de autenticación personalizada, lo cual permite la creación de un sistema por código.

Las soluciones empleadas fueron las de Google: la familia de productos *Firebase*.

Un aspecto a comentar es que, para generar un token, es necesario un identificador de usuario. Aunque, como se ha mencionado antes, la skill proporciona uno, este es demasiado largo para el servicio de autenticación utilizado. Por otro lado, la base de datos genera un identificador más corto por cada documento creado con los datos de un usuario. Es por ello que se utiliza este último para generar los *tokens*. Debido a esto, la *API* que valida los códigos ha de realizar una llamada extra a la base de datos para "traducir" el identificador (es decir, obtener la versión corta a partir de la larga).

4.2 Diseño Detallado

4.2.1. Estructura de directorios

Tipo	Raíz[7]	Recurso	Contenido
Configuración	/	.ask/ ask-resources.json	Datos para el ASK CLI (generados por éste)
Skill package	/skill-package	skill.json	*Endpoints*[8] de las APIs Nombre e icono de la skill
		assets/	Otros recursos (imágenes)
	interactionModels/custom	*xx-YY*.json[9]	Modelo de interacción (distinto en cada idioma)
Código	/	lambda/	Código fuente

Tabla 4.1: Estructura de directorios en un proyecto de skill

[5] https://aws.amazon.com/es/cognito/

[6] https://firebase.google.com/products/auth?hl=es

[7] Carpeta respecto a la que las rutas de cada recurso son relativas. Si la ruta de una carpeta "raíz" no es absoluta, significa que es relativa a la carpeta de la celda superior.

[8] En este caso, un *endpoint* es una dirección a la que Alexa debe llamar para invocar a la skill.

[9] El nombre de estos archivos indica mediante códigos el idioma *(xx) (ISO 639-1)* y país *(YY) (ISO 3166-1)* al que el modelo de interacción está asociado. (Ejemplo: *"es-ES.json"* sería la versión en español de España)

En un proyecto de skill, existen tres principales grupos de recursos (tabla 4.1): Código, configuración, y *skill package* (traducido al español sería *"paquete de la skill"*).

Los dos primeros grupos no requieren mucha explicación, ya que sus propios nombres dan información de sobra. Sin embargo, el *skill package* sí. Este grupo engloba: archivos con metadatos sobre la skill (como pueden ser el nombre e icono); otros recursos como las imágenes asociadas; y el *modelo de interacción*, el cual representa la interfaz de la skill.

4.2.2. Lenguaje de programación

Antes de entrar en la lógica de la aplicación, es importante comentar el lenguaje de programación que se ha elegido para desarrollar la skill.

JavaScript es uno de los lenguajes de programación con soporte en las *Alexa-hosted* skills (concepto explicado en el capítulo 2), y es el lenguaje en el que se encuentra el código con el cual Alexa realiza el despliegue de la skill "Al Loro". Sin embargo, no se trata del lenguaje en el que se ha desarrollado realmente la skill. ¿Cómo es esto posible?

TypeScript es un lenguaje desarrollado por *Microsoft* que *extiende* la funcionalidad de *JavaScript*. Esto quiere decir que, cuando se escribe código en *TypeScript*, este pasa por un proceso de *compilación*[10] en el que se traduce a código *JavaScript*. Este ha sido el lenguaje utilizado para desarrollar "Al Loro", y la razón de que una situación así sea posible.

Las principal ventaja de utilizar *TypeScript* sobre *JavaScript* es que el primero añade un sistema de tipos del que el segundo carece, proporcionando robustez y un mayor nivel de mantenibilidad al código. Además, proporciona *intellisense* (la capacidad del editor de código de mostrar documentación y definiciones de las diferentes funciones de una librería), lo cual facilita mucho el proceso de desarrollo.

4.2.3. Lógica

Handlers

La lógica de una skill es muy similar a la de una API en desarrollo backend (lo cual se debe a que internamente se trata de una). Esta sigue un patrón petición-respuesta, en el que las peticiones (provenientes de Alexa) incluyen el estado de la conversación y el intent que considera que el usuario ha invocado. El trabajo del desarrollador es el de implementar ***handlers***[11], segmentos de código que se encargan de reaccionar a cada evento (puede ser un *intent*, un *error*, o de otro tipo, como el *lanzamiento* o *finalización* de una sesión) y producir una respuesta adecuada.

Por suerte, Amazon proporciona los *ASK SDK*[12], una serie de herramientas y librerías que facilitan el trabajo a los desarrolladores al abstraer la implementación interna, ofreciendo opciones más simples y directas de acceder a las diferentes funcionalidades de Alexa.

El archivo principal (*index.ts*)[13] contiene el siguiente segmento de código:

[10] Técnicamente, el compilador de *TypeScript* se considera un *transpilador*, debido a que tanto el lenguaje de origen como de destino son considerados de alto nivel (es decir, tienen un nivel de abstracción similar).

[11] https://developer.amazon.com/es-ES/docs/alexa/alexa-skills-kit-sdk-for-nodejs/handle-requests.html#request-handlers

[12] https://developer.amazon.com/es-ES/docs/alexa/sdk/alexa-skills-kit-sdks.html

[13] El nombre del archivo es importante porque, en una skill escrita en JavaScript, el archivo principal debe llamarse '*index.js*' y encontrarse en la carpeta raíz de la sección de código del proyecto. Como "Al Loro" está desarrollado en TypeScript, debe llamarse '*index.ts*' y encontrarse en la misma carpeta.

```
export const handler = SkillBuilders.custom()
    // Esta id de skill es un ejemplo con el mismo formato
    .withSkillId('amzn1.ask.skill.9cfe0c01-4d39-fcbe-b66e-be36dfb7c43b')
    .addRequestHandlers(
        LaunchRequestHandler,
        CancelAndStopIntentHandler,
        HelpIntentHandler,
        AuthIntentHandler,
        ReadIntentHandler,
        ReadItemIntentHandler,
        ReadContentIntentHandler,
        GoToPreviousItemIntentHandler,
        SkipItemIntentHandler,
        ListIntentHandler,
        SessionEndedRequestHandler,
        RepeatIntentHandler,
        IntentReflectorHandler
    )
    .addRequestInterceptors(LocalizationRequestInterceptor)
    .addResponseInterceptors(SaveResponseForRepeatingInterceptor)
    .addErrorHandlers(
        InvalidFeedUrlErrorHandler,
        FeedIsTooLongErrorHandler,
        GenericErrorHandler
    )
    .withPersistenceAdapter(new FirebaseWithDynamoDbPersistenceAdapter())
    .lambda();
```

Este segmento representa el punto de entrada de la skill, puesto que esta serie de llamadas generan una función que *AWS Lambda* espera encontrar en una variable llamada "*handler*" y utilizará para enrutar todas las peticiones a los diferentes *handlers* específicos que se proporcionen.

Un *handler* está formado por dos funciones con las siguientes firmas:

```
canHandle(input: HandlerInput): boolean | Promise<boolean>;
handle(input: HandlerInput): Response | Promise<Response>;
```

- *canHandle* recibe los datos de entrada de la petición, y debe devolver un booleano indicando si su función *handle* es capaz de manejar el evento en base a la petición recibida.
- *handle* es el encargado de realizar las acciones asociadas a la petición recibida como entrada, y de generar una respuesta adecuada.

Interceptors

Por otro lado se encuentran los ***interceptors***[14]: se tratan de objetos con una única función (de nombre *process*) que intercepta los datos en algún punto del proceso de petición y respuesta y realiza alguna acción en base a su estado. No se espera que esta función retorne ningún valor. Existen dos clases de estos objetos:

- *RequestInterceptor*: Se ejecuta en cuanto se recibe una petición, pero antes de que esta sea manejada.
- *ResponseInterceptor*: Se ejecuta tras obtener una respuesta, pero antes de mandarla de vuelta al emisor.

[14] https://developer.amazon.com/es-ES/docs/alexa/alexa-skills-kit-sdk-for-nodejs/handle-requests.html#request-and-response-interceptors

Directives

Las **directivas**[15] (*directives* en inglés) son instrucciones especiales que indican a Alexa un cambio en el estado del diálogo.

Existen dos principales tipos de directivas:

- **Delegación del diálogo a Alexa**: Esta categoría está formada por una única directiva: *Delegate*. Básicamente, es una forma de ceder el control del diálogo a Alexa, permitiendo que continúe con el siguiente paso del mismo. Por defecto, el control es delegado automáticamente; sin embargo, esta opción se puede deshabilitar, o se puede controlar manualmente en el código mediante otra directiva.

- **Control manual del diálogo**: Estas directivas permiten un control más preciso sobre el diálogo, ordenando a Alexa que le pida al usuario ciertas cosas dentro del mismo. Se tratan de:

Nombre	**Petición**
ElicitSlot	El valor de un *slot* concreto
ConfirmSlot	Confirmación para aceptar el valor del *slot* recibido
ConfirmIntent	Confirmación para realizar un *intent* concreto

Estas directivas pueden también aplicar cambios directamente a elementos del diálogo como el *intent* o el valor actual de un *slot*.

En el caso de "Al Loro", se utilizan dos de ellas:

- *Delegate* se usa en dos estados:

  - Cuando el usuario rechaza el intent asociado a seguir leyendo un contenido que se ha debido truncar en varios segmentos por ser muy largo, se pasa automáticamente al siguiente elemento de la feed cambiando el intent para que esto suceda.
  - Cuando el usuario rechaza la confirmación de pasar al siguiente elemento, se actualiza el intent al de "cancelar".

    (Esta confirmación ocurre al acabar de leer un elemento, con la idea de evitar un error en el diálogo: si el usuario pedía pasar al siguiente mientras está leyendo el contenido de un elemento, se saltaba un elemento por completo. Esto ocurría porque al leer un contenido, el contador interno ya había pasado al siguiente, porque después de un contenido leía el siguiente título.)

- *ConfirmIntent* se utiliza para pedir al usuario confirmación para la siguiente acción tras leer un segmento del contenido de un elemento de una feed. Puede ser una de dos:

  - Si queda contenido por leer, pregunta si quiere seguir escuchando.
  - Si ya ha terminado de leer todo el contenido, pregunta si quiere pasar al siguiente elemento.

[15] https://developer.amazon.com/es-ES/docs/alexa/custom-skills/dialog-interface-reference.html

Además, existe una directiva que no encaja con ninguna de las categorías mencionadas: *UpdateDynamicEntities*. Esta directiva permite añadir valores dinámicos a los slots personalizados, para poder adaptar el comportamiento de la skill a ciertos estados.

Un claro ejemplo de su utilidad se puede observar en la skill "Al Loro" a la hora de que el usuario quiera indicar el nombre de la feed que desea escuchar. Sin esta funcionalidad, los nombres de las feeds deberían estar indicados por el desarrollador en el modelo de interacción para que fueran correctamente resueltos por la skill. Por tanto, los valores de esta lista estática serían los únicos permitidos, y el usuario se vería obligado a escoger un nombre de esta para su feed en lugar de poder añadir uno propio. En cambio, con la directiva *UpdateDynamicEntities* se pueden añadir los nombres de las feeds del usuario en tiempo de ejecución, cada vez que se inicie la skill.

4.2.4. Persistencia

Para abstraer la persistencia de la lógica, en Alexa hay un concepto llamado "**atributos**"[16] (*attributes*). Básicamente consisten en diccionarios clave-valor, donde la clave es una cadena de texto (*String*), y el valor es un objeto. Existen tres tipos de atributos, clasificados según su esperanza de vida:

- ***RequestAttributes***: Este tipo de atributos es descartado tras cada petición. Por este motivo, no requieren ningún tipo de serialización[17], ya que no necesitan ser enviados a ningún servicio externo a la lógica de la skill (al contrario que los otros dos). Su principal utilidad es en conjunción con los *interceptors*, ya que estos pueden incluir objetos, funciones o datos que puedan ser utilizados por los *handlers*.
- ***SessionAttributes***: Son probablemente los más importantes, y se mantienen a lo largo de una sesión. Vienen incluidos en cada petición y se actualizan con cada respuesta. Esto último acarrea una importante limitación (5.6.1). Son útiles para guardar el estado de una conversación.
- ***PersistentAttributes***: Como su nombre indica, estos atributos persisten más allá de la sesión en la que son creados. Son los más complejos, pues dependen de una interfaz llamada *PersistenceAdapter*, cuya implementación define la integración con la base de datos.

Estos atributos pueden ser gestionados a través de las funciones del objeto *AttributesManager*, incluido en la petición recibida por cada handler.

PersistenceAdapter

Para facilitar la persistencia de datos entre sesiones, el ASK SDK proporciona una interfaz llamada *PersistenceAdapter*. Esta contiene tres funciones que reciben como entrada un objeto de tipo *RequestEnvelope* (el cual representa la petición completa), y realizan operaciones relacionadas con la persistencia. Estas son:

- *getAttributes*: Devuelve todos los atributos que se encuentran persistidos.

[16] https://developer.amazon.com/es-ES/docs/alexa/alexa-skills-kit-sdk-for-nodejs/manage-attributes.html

[17] En el caso de Alexa, la *serialización* consiste en una conversión de cada objeto a una cadena de texto con formato *JSON*, que luego se empaqueta en cada respuesta. Por ello, no es recomendable que estos objetos contengan funciones (ya que el formato no las soporta y se pierden).

- *saveAttributes*: Además del *RequestEnvelope*, recibe un diccionario con atributos que esta función persiste.
- *deleteAttributes* (opcional): Elimina todos los atributos existentes en el sistema de persistencia.

Para hacer uso de esta interfaz, uno debe crear una nueva instancia de un objeto perteneciente a una clase que la extienda, y añadirla al handler del punto de entrada de la skill con la llamada "*.withPersistenceAdapter(adapter)*" (siendo "*adapter*" una variable con dicha instancia).

```
1    .withPersistenceAdapter(new FirebaseWithDynamoDbPersistenceAdapter())
```

Amazon proporciona librerías que contienen este tipo de clases con implementaciones para los servicios de persistencia que incluye la skill[18] (actualmente *S3* y *DynamoDB*). Para otro tipo de sistemas (como puede ser *Firestore*), uno puede crear su propia implementación.

Finalmente se hizo esto último en "Al Loro", pero no fue la primera opción.

4.2.5. Base de datos

Los dos sistemas de base de datos de los que la skill "Al Loro" hace uso siguen el modelo no relacional (*NoSQL*). La principal diferencia respecto al clásico modelo relacional (*SQL*) es que no requiere de un esquema fijo para los datos que se almacenan en él. Esto aporta una mayor flexibilidad y rapidez, especialmente en etapas tempranas de desarrollo, pues no es necesario que el desarrollador tenga que saber la estructura exacta que tendrán sus datos mientras va probando.

En ambos sistemas, los datos están representados en su capa más baja como *campos* de pares clave-valor. Cada valor puede tener un *tipo* distinto (cadena de texto, numérico, booleano...), pero este debe estar definido dentro del campo.

Firebase (Firestore)

En Firestore, los datos se almacenan del siguiente modo:

- En la capa más alta, se encuentran las *colecciones*, listas que dividen los datos según a qué están asociados. El equivalente en *SQL* serían las *tablas*.
- Dentro de cada colección se ubican los *documentos*. El equivalente en *SQL* serían las *filas* que componen una tabla, y cada uno representa una unidad del objeto al que hacen referencia.
- Dentro de cada documento, se encuentran los diferentes *campos*.
- Opcionalmente, un documento también puede contener una o varias *subcolecciones*. La única diferencia con las colecciones es que se encuentran en una capa inferior.

Cada una de las estructuras comentadas contiene un identificador (*ID*), que es generado automáticamente en el momento de su creación.

[18] Lista de librerías que forman parte del *ASK SDK* para *Node.js*:
https://github.com/alexa/alexa-skills-kit-sdk-for-nodejs#package-versions

Un aspecto interesante de Firestore son sus *reglas de seguridad*[19] (figura 4.2), un sistema de control de acceso donde uno puede definir las condiciones que otorgan acceso a cada uno de los recursos de la base de datos.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId}/feeds/{document=**} {
      allow read, write: if request.auth != null && request.auth.uid == userId
    }
    match /{document=**} {
      allow read, write: if false;
    }
  }
}

```

Figura 4.2: Reglas de seguridad de Firestore en "Al Loro"

La sintaxis de las reglas es relativamente sencilla:

- La directiva *match* recibe una ruta en forma de patrón. Los recursos con los que coincida serán aquellos afectados por las reglas que se incluyan dentro de ésta (es decir, entre las llaves) Además, las partes de la ruta entre llaves podrán utilizarse como variables mediante el nombre indicado.
- La directiva *allow* sigue la siguiente estructura: *allow <permiso 1, permiso 2...>: if <condición>*, donde los permisos disponibles están divididos como:
  - Lectura (*read*)
    - Obtener (*get*)
    - Enumerar (*list*)
  - Escritura (*write*)
    - Crear (*create*)
    - Actualizar (*update*)
    - Eliminar (*delete*)
- La existencia (o no) del objeto *request.auth* determina si el usuario está autenticado. En el caso de que lo esté, se puede obtener su identificador de usuario mediante el atributo *uid*. Esto puede servir para restringir el acceso a los datos de un usuario, dando permiso únicamente a un usuario autenticado con la misma *uid*.

DynamoDB

En la base de datos DynamoDB, las cosas son un poco diferentes:

- Al igual que en *SQL*, las listas de la capa más alta se denominan *tablas*.
- Las estructuras análogas a los *documentos* de Firestore tienen por nombre *elementos* (*items* en inglés), y los campos se llaman *atributos*.

[19] https://firebase.google.com/docs/firestore/security/get-started?hl=en

Al contrario que en Firestore, los identificadores deben ser procurados por el desarrollador, mediante la definición de aquellos atributos que vayan a realizar dicha función.

El identificador puede ser de dos tipos:

- *Partition key*: Compuesta por un único atributo, indica el valor que DynamoDB utilizará para crear una clave *hash* que determine en qué partición se almacenará internamente el elemento.
- *Partition key* y *Sort key*: Similar al anterior, este tipo de identificador también incluye un segundo atributo que define cómo ordenará cada elemento DynamoDB dentro de una misma partición.

El ejemplo que da la *guía del desarrollador de DynamoDB*[20] para comprender mejor estos conceptos es el de una tabla llamada "música", donde existen dos atributos: "artista", que sería la *partition key*; y "título de la canción", que representaría la *sort key*. La ventaja de tener un esquema así es que, si uno realiza una consulta de todas las canciones de un determinado artista, DynamoDB será más rápido en ofrecer una respuesta, pues todas ellas se encontrarán físicamente en la misma partición.

[20] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html

CAPÍTULO 5

Desarrollo de la solución propuesta

5.1 Gestión del proyecto

Para realizar la gestión del proyecto se apostó por la plataforma ***Trello***[1], puesto que permite la fácil creación de tableros *kanban*; un sistema de organización basado en **listas** y **tarjetas**. Se puede ver un ejemplo de ello en la figura 5.1.

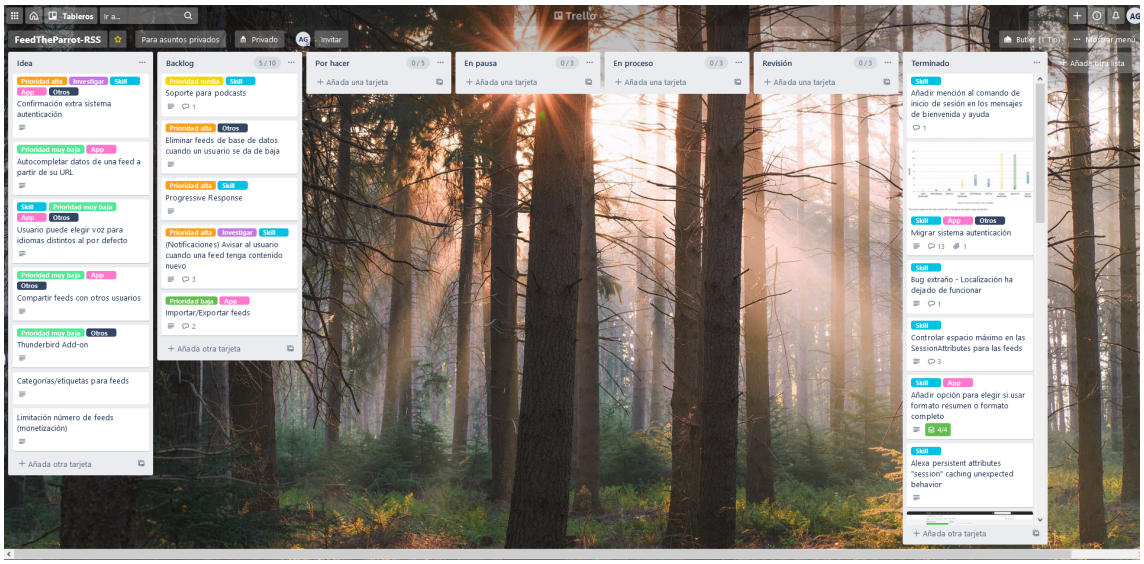


Figura 5.1: Captura de pantalla del tablero kanban en la web de Trello.

Cada **tarjeta** representa una tarea, y se utilizaron **etiquetas** para clasificarlas según su:

- **Prioridad**: muy alta, alta, media, baja, muy baja.
- **Plataforma**: skill, app, otros.

Además, se incluyó una etiqueta más: **investigar**, indicativa de que va a hacer falta una investigación más exhaustiva de lo habitual para poder ser completada.

Las **listas** simbolizan, en este caso, el proceso de desarrollo de una tarea. Los estados que se incluyeron son:

[1] https://trello.com/

- **Idea**: La tarea no es más que un concepto. No tiene por qué estar claramente especificada, pues puede haber surgido de un proceso de búsqueda de ideas. Aun no se sabe si se hará, y puede ser rechazada en cualquier momento.
- **Backlog**: La idea ya ha sido concretada y categorizada mediante las etiquetas. En principio se espera que se acabe completando, pero no se sabe cuándo.
- **Por hacer**: Se confirma que la tarea debe ser realizada en un futuro cercano.
- **En pausa**: La tarea se encontraba en proceso, pero su desarrollo ha tenido que ser detenido. Esto podría ocurrir porque ha surgido otra tarea con mayor urgencia, o debido a un contratiempo.
- **En proceso**: La tarea se está realizando.
- **Revisión**: En principio la tarea se ha completado, pero debe comprobarse que todo funciona correctamente antes de darla por terminada.
- **Terminado**: El desarrollo de la tarea ha finalizado.

5.2 Primeros pasos

5.2.1. Aprendizaje

Previo al desarrollo de la skill, hubo un proceso inicial de aprendizaje para saber cómo construir una skill desde cero; y un recurso muy útil para ello fue la serie de tutoriales "De Cero a Héroe" ("*Zero to Hero*") del canal oficial de *Alexa Developers* en *YouTube*.[2] En un principio, se hizo uso del contenido en inglés[3] para dar los primeros pasos. Más adelante, se descubrió la existencia de una versión en español[4]; creada unos meses después, y con algunos vídeos más.

En estos tutoriales se explican con detalle algunos conceptos importantes en el desarrollo de skills, como: la *internacionalización*, los *interceptors* (4.2.3), los *slots* (5.3), o la *persistencia* (4.2.4), entre otros.

5.2.2. Proceso de creación de una skill

Actualmente, el proceso necesario para crear una skill por primera vez sería el siguiente:

1. Iniciar sesión con una cuenta de Amazon para acceder a la consola de desarrollador de Alexa.[5]
2. Pulsar el botón "*Create Skill*" para comenzar el proceso de creación de una skill. (figura 5.2)
3. Introducir un nombre para la skill (de entre 2 y 50 caracteres).
4. Seleccionar el idioma por defecto (podrán añadirse más idiomas a la skill tras terminar el proceso de creación).[6]

[2] https://www.youtube.com/c/AlexaDevelopers
[3] En inglés: https://youtube.com/playlist?list=PL2KJmkHeYQTO65ko4I--OC-7CC_Cjg8sS
[4] En español: https://youtube.com/playlist?list=PL2KJmkHeYQTNzra-T2ayhV_84dHYCShAQ
[5] https://developer.amazon.com/alexa/console/ask?
[6] Actualmente, los idiomas disponibles son: inglés (US, UK, AU, CA, IN), español (ES, US, MX), alemán (DE), francés (FR, CA), italiano (IT), japonés (JP), portugués (BR), hindi (IN).

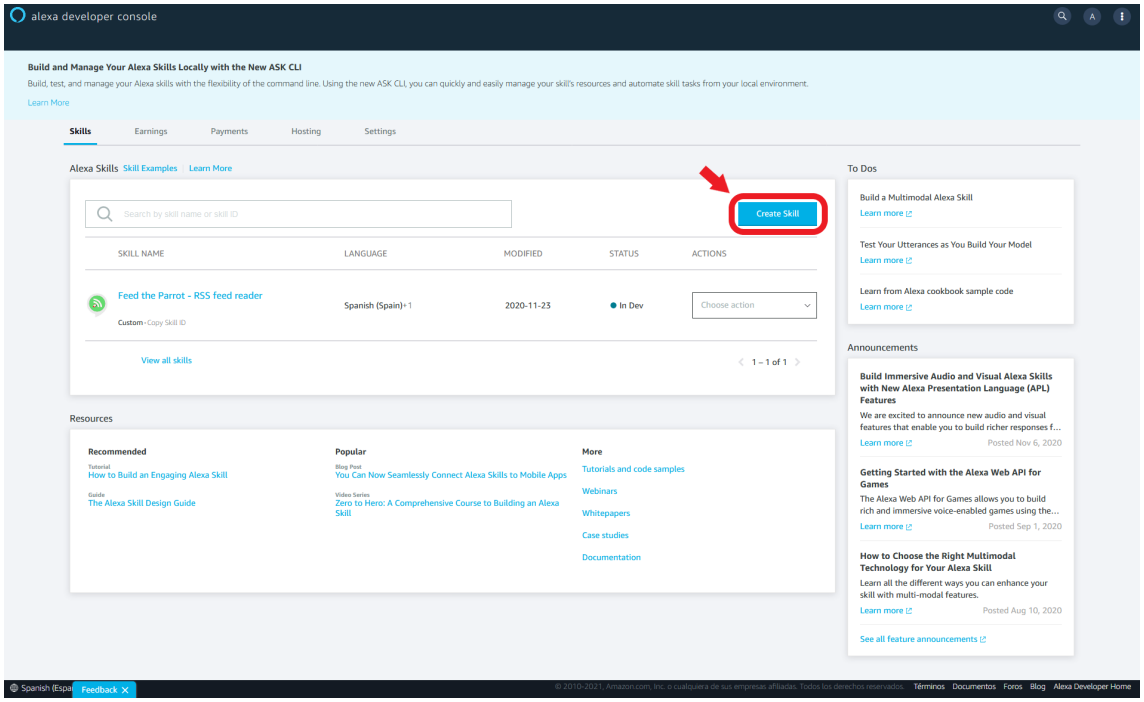


Figura 5.2: Pantalla inicial de la consola de desarrollador de Alexa, resaltando el botón para crear una skill.

5. Elegir un modelo para la skill, o diseñar uno propio.

   Alexa dispone de varios modelos de skill pre-construidos para desarrollar: noticias, *smart home* (domótica), o vídeo (figura 5.3). También puedes diseñar tu propio modelo personalizado.

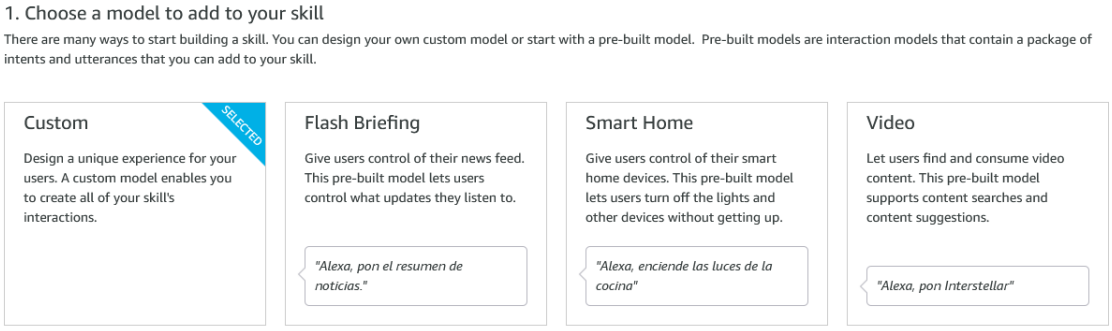


Figura 5.3: Selección de modelo en la pantalla de creación de una skill.

6. Escoger un método para hospedar los recursos de backend de la skill (figura 5.4).

   El método por defecto se llama "*Alexa-hosted*" [6], y consiste en que Alexa maneja todos los recursos por ti. Te proporciona acceso al servicio *AWS Lambda* automáticamente (este es el servicio que hospedará el código que se ejecutará cuando se use la skill), además de un servicio de base de datos (*DynamoDB*[7]), uno de almacenamiento de archivos multimedia (*Amazon S3*[8]), y un repositorio de código (*AWS CodeCommit*[9]).

[7] https://aws.amazon.com/es/dynamodb/
[8] https://aws.amazon.com/es/s3/
[9] https://aws.amazon.com/es/codecommit/

Este método está disponible para los lenguajes: *JavaScript* (mediante el entorno *Node.js*), y *Python*.

Los aspectos a tener en cuenta de este método son:

- Por un lado, tendrás acceso a un editor de código directamente en la web de la consola de desarrollo.
- Pero por otro, tanto la personalización de estos servicios como su uso están limitados a la capa gratuita de *AWS*.

Por ello, si la skill necesita muchos recursos, o el desarrollador desea un mayor control sobre estos, existe la alternativa de proveerlos por su cuenta.

En cualquier caso, el primer método es el más recomendable en la amplia mayoría de situaciones: es el más rápido, sencillo, y el que ofrece las mayores ventajas desde el primer momento. Además, se encuentra en constante evolución y mejora.

Asimismo, en el caso de que se quiera integrar la skill con otros servicios de *AWS*, existe una opción para integrar una cuenta de *AWS* con una skill hospedada por Alexa.

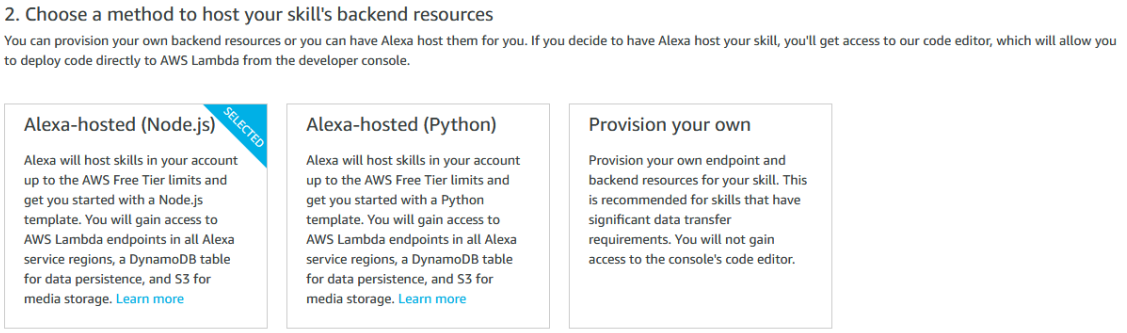


Figura 5.4: Selección de método de hospedaje (*hosting*) en la pantalla de creación de una skill.

7. Opcionalmente, se puede cambiar la región en la que se hospedan los recursos de la skill.

8. Pulsar el botón "*Create skill*" para terminar el proceso y acceder a la consola.

5.3 Interfaz - Modelo de Interacción

En Alexa existe un concepto importante a la hora de comprender en qué consiste un modelo de interacción: los *intents*. La traducción al español de esta palabra sería "intenciones", y esto es exactamente lo que simbolizan: la intención del usuario en cada momento.

Un intent está formado por:

- Un nombre que lo identifique
- Una lista de *samples* (muestras) de frases que puede decir un usuario para desencadenar dicho intent
- Opcionalmente, una lista de *slots* ("ranuras") con una serie de variables que puedan venir incluidas en la frase.

Un ejemplo de un intent con slots en la skill de "Al Loro" sería el de leer una feed (figura 5.5). Para ello, un usuario podría decir: "Lee la feed *"noticias de la ETSINF""* (donde *"noticias de la ETSINF"* sería un slot con el nombre de una feed).

```
{
   "name": "ReadIntent",
   "slots": [
     {
       "name": "feed",
       "type": "FeedName",
       "samples": [
         "Se trata de la feed {feed}",
         "Se trata de {feed}",
         "Es {feed}",
         "Es la feed {feed}",
         "La feed se llama {feed}",
         "Se llama {feed}",
         "{feed}"
       ]
     }
   ],
   "samples": [
     "que lea la feed {feed}",
     "que lea una feed",
     "léeme la feed {feed}",
     "lee la feed {feed}",
     "léeme una feed",
     "lee una feed"
   ]
}
```

Figura 5.5: Extracto de código JSON de un modelo de interacción en español para el intent "ReadIntent", que representa la intención de que se lea una feed

En el caso de los slots, el usuario no tiene por qué acordarse de nombrarlos cuando recita la frase. Es por ello que existe una opción para activar la *"**elicitation**"*: un proceso mediante el cual se obtienen los slots a través de una serie de pequeños mensajes (también indicados por el propio desarrollador) en los que Alexa pregunte al usuario por cada uno de ellos (una posible pregunta sería: "¿Cómo se llama la feed que debo leer?"). Del mismo modo, deben añadirse muestras de posibles respuestas por parte del usuario, como: "Es la feed *{feed}*" o "Se trata de la feed *{feed}*".

Es importante mencionar que no todos los intents deben ser creados desde cero por el desarrollador; los más comunes son proporcionados por Amazon, y son llamados ***built-in* intents** (intents incorporados). Sus nombres vienen siempre precedidos por "*AMAZON.*" para diferenciarlos de los personalizados. Ejemplos de este tipo de intents son aquellos para decir:

- "Sí" (*AMAZON.YesIntent*)
- "No" (*AMAZON.NoIntent*)
- "Repetir" (*AMAZON.RepeatIntent*)

Algunos de ellos también son de uso obligatorio, como:

- "Cancelar" (*AMAZON.CancelIntent*)
- "Parar" (*AMAZON.StopIntent*)
- "Ayuda" (*AMAZON.HelpIntent*)

5.4 Internacionalización

5.4.1. Skill

La primera noción en los tutoriales relacionada con la lógica de la skill es, curiosamente, la ***internacionalización***. Es muy posible que esto se deba a que una skill puede estar disponible en varios lenguajes, además del interés que parece tener la empresa detrás de Alexa en la globalización y el soporte en múltiples idiomas (cosa que da a entender el hecho de que estos tutoriales estén disponibles en más de una lengua). Para "Al Loro", se decidió dar soporte al **inglés** (de Estados Unidos) y al **español** (de España).

Para saber el idioma del dispositivo en cada sesión, el ASK SDK proporciona una función de utilidad llamada "*getLocale*", el cual recibe un objeto de tipo *RequestEnvelope* (el cual viene incluido en cada petición) y devuelve el idioma como una cadena de texto.

Librería utilizada

Sabiendo el idioma, el siguiente paso es tener un sistema que, en base al idioma y algún tipo de clave, sea capaz de devolver cada mensaje en el lenguaje adecuado. La librería que se presenta para lograr esta funcionalidad es "*i18next*"[10].

Esta librería (en su forma más básica, pues contiene una serie de *plugins* que pueden expandir su funcionalidad) utiliza un objeto con la siguiente estructura para cargar los textos en diferentes idiomas:

- L1
  - E1
    - C1: V1
    - C2: V2
  - E2
    - C3: V3
    - C4: V4
- L2
  - E1
    - C1: V5
    - C2: V6
  - E2
    - C3: V7
    - C4: V8

[10] https://www.npmjs.com/package/i18next

Las letras representan:

- (L) -> Lenguaje
- (E) -> Espacio de nombres (*namespace*)
- (C: V) -> Clave: Valor

La idea es que cada texto tendrá una misma clave para todos los idiomas en los que se encuentre.

El espacio de nombres sirve para agrupar las claves. Por defecto, el espacio de nombres es "*translation*", y salvo en proyectos más complejos, es normalmente el único que se usa.

Un ejemplo práctico de este tipo de objeto es el siguiente:

```
const miObjeto = {
   en: {
      translation: {
         "HELLO_MSG": "Hello"
      }
   },
   es: {
      translation: {
        "HELLO_MSG": "Hola"
      }
   }
}
```

Para utilizar este sistema se utiliza una función de inicio, al que se le pasan el idioma del dispositivo y un objeto con la estructura anterior, el cual devuelve otra función que hace de traductor.

A esta nueva función traductora se le pasa una clave, y devuelve el valor asociado al idioma que se indicó en el momento de su creación.

```
const traduce = await i18next.init({
   lng: 'es',
   resources: miObjeto
})

traduce("HELLO_MSG") // "Hola"
```

Un aspecto a tener en cuenta es que, cuando se realiza la llamada a *init* directamente, el objeto creado se mantiene como un *singleton*: esto implica, por un lado, que uno podría llamar a la función aquí llamada "*traduce*" del siguiente modo (y también funcionaría):

```
i18next.t("HELLO_MSG") // "Hola"
```

Pero también quiere decir que si uno realizara una segunda llamada a la función *init*, el primer objeto sería reemplazado por el segundo, lo cual puede llevar a errores inesperados. La alternativa a este método sería encadenar la llamada a *init* con una llamada previa a la función "*createInstance*", pasando ahora los argumentos sólo a esta última:

```
const traduce = await i18next.createInstance({
   lng: 'es',
   resources: miObjeto
}).init()

traduce("HELLO_MSG") // "Hola"
i18next.t("HELLO_MSG") // Error
```

De esta forma, con cada inicialización se crean diferentes instancias. Aunque es posible que sea menos eficiente en recursos, también es más versátil y menos propensa a errores, especialmente teniendo en cuenta el modo en el que deben integrarse con los handlers en el desarrollo de skills.

Integración con los handlers

Un aspecto importante en el desarrollo de una *API* es que, salvo que se persista de algún modo, cada petición funciona como una ejecución del programa independiente del resto. Esto quiere decir que, cada vez que se recibe una petición, se debe partir de cero.

Este funcionamiento puede chocar un poco con el modelo de internacionalización comentado, pues se basa en tener una función traductora que se mantiene durante toda la ejecución. En un principio, se podría pensar en persistir esta función durante toda la sesión. Sin embargo, debido al funcionamiento interno de la persistencia en Alexa, las funciones no pueden ser persistidas. Por tanto, debe crearse una nueva instancia por cada petición que se reciba.

Por otro lado, el sistema de internacionalización debería ser independiente del handler que maneje cada petición; pues, en principio, lo ideal sería que la función estuviese disponible en todos. Para conseguir esto, se puede hacer uso de un *interceptor* (4.2.3) (concretamente, un *RequestInterceptor*).

```
export const LocalizationRequestInterceptor: RequestInterceptor = {
    async process(handlerInput) {
        const t = await initNewInstance(getLocale(handlerInput.requestEnvelope));
        const attributes = handlerInput.attributesManager.getRequestAttributes();
        attributes.t = function (...args: any[]) {
            return (<any>t)(...args);
        };
    },
};
```

Este segmento de código se ejecutará antes de cada handler, creando una nueva función traductora cada vez y guardándola en los *RequestAttributes*, a los que el handler tiene acceso.

Responder en múltiples idiomas

Un caso especial que no contempla el tutorial oficial es el de **responder en un idioma distinto al del dispositivo**. Aunque esto puede parecer una nimiedad, es más complejo de lo que parece, ya que cada voz en Alexa está asociada a un idioma concreto, y la forma en que pronuncia las palabras está relacionada con este. En la práctica, esto lleva a que expresiones en otros idiomas suenan como si las pronunciase alguien no nativo[11].

En el caso concreto de "Al Loro", resultaba útil la idea de poder tener feeds en otros idiomas y poder escucharlas aunque el idioma de la feed y el del dispositivo no concuerden.

Para llevar esta idea a la práctica, es necesario conocer un lenguaje especial de marcado que se puede utilizar para indicar a Alexa cómo debe expresar las respuestas orales: *SSML*[12] (*Speech Synthesis Markup Language*).

[11] Más información en: https://developer.amazon.com/es-ES/docs/alexa/custom-skills/speech-synthesis-markup-language-ssml-reference.html#lang

[12] https://developer.amazon.com/es-ES/docs/alexa/custom-skills/speech-synthesis-markup-language-ssml-reference.html

Este lenguaje contiene una serie de etiquetas (*tags*) que permiten: introducir pausas, añadir énfasis, modificar el volumen o velocidad, indicar la pronunciación, cambiar el idioma o la voz... entre otras cosas.

Para lograr el efecto deseado, la mejor opción es combinar las etiquetas "*voice*" y "*lang*" para cambiar la voz y el idioma, respectivamente. Un ejemplo de cómo se conseguiría esto es:

```
<voice name="Kendra"><lang xml:lang="en-US">Example</lang></voice>
```

Los nombres de las voces provienen de una lista provista por Amazon[13].

5.4.2. App

Para mantener una consistencia en la disponibilidad de idiomas entre la skill y la app, se decidió (más adelante en el desarrollo) implementar un sistema de internacionalización también en la app.

Por desgracia, la documentación al respecto[14] ha cambiado tras el reciente lanzamiento de la versión 2 de Flutter[15]. Por tanto, el método que se muestra ahí podría variar respecto al que se utilizó en su momento (que es el que se explicará en esta sección).

Traducciones automáticas

El texto que la plataforma Flutter genera automáticamente se encuentra por defecto en inglés (Estados Unidos). Para poder tener el texto traducido al español, se debe importar la librería *flutter_localizations*. Una vez hecho esto, se deben añadir las clases asociadas a la internacionalización como parte de un argumento en la llamada al constructor de la clase principal de la app (*MaterialApp*), así como los idiomas soportados por la app.

```
1  MaterialApp(
2    // ...
3    localizationsDelegates: [
4      GlobalMaterialLocalizations.delegate,
5      GlobalWidgetsLocalizations.delegate,
6      GlobalCupertinoLocalizations.delegate,
7    ],
8    supportedLocales: [const Locale('en'), const Locale('es')],
9    // ...
10 )
```

Mensajes personalizados

Para añadir mensajes personalizados en diferentes idiomas, el sistema es más complejo.

En primer lugar, uno debe crear una clase con un segmento de código que es siempre igual (*boilerplate code*):

[13] https://developer.amazon.com/es-ES/docs/alexa/custom-skills/speech-synthesis-markup-language-ssml-reference.html#supported-voices

[14] https://flutter.dev/docs/development/accessibility-and-localization/internationalization

[15] https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65

```
class AppLocalizations {
   AppLocalizations(this.localeName);

   static Future<AppLocalizations> load(Locale locale) {
      final String name = locale.countryCode == null || locale.countryCode.
    isEmpty
          ? locale.languageCode
          : locale.toLanguageTag();
      final String localeName = Intl.canonicalizedLocale(name);
      return initializeMessages(localeName).then((_) {
         return AppLocalizations(localeName);
      });
   }

   static AppLocalizations of(BuildContext context) {
      return Localizations.of<AppLocalizations>(context, AppLocalizations);
   }

   final String localeName;

   // ...
}
```

Tras este código, se van creando *getters* (funciones que se comportan como variables de sólo lectura) para cada mensaje, con la siguiente estructura:

```
class AppLocalizations {

   // ...

   String get appTitle {
      return Intl.message(
         'Feed the Parrot', // Mensaje sin traducir
         name: 'appTitle', // Nombre asociado al mensaje
         desc: 'Title for the application', // Descripción del mensaje
         locale: localeName, // Esto es siempre igual
      );
   }
}
```

A continuación, se crea una clase más con el siguiente código (la mayoría, de nuevo, siempre igual):

```
class AppLocalizationsDelegate extends LocalizationsDelegate<AppLocalizations> {
  const AppLocalizationsDelegate();

  // Aquí se indican los idiomas soportados
  @override
  bool isSupported(Locale locale) => ['en', 'es'].contains(locale.languageCode)
    ;

  @override
  Future<AppLocalizations> load(Locale locale) => AppLocalizations.load(locale)
    ;

  @override
  bool shouldReload(AppLocalizationsDelegate old) => false;
}
```

De esta última clase se crea una instancia que se añade como argumento al constructor de *MaterialApp* (similar a lo que ocurría en la sección anterior):

```
   MaterialApp(
```

```
    localizationsDelegates: [
      const AppLocalizationsDelegate(),
    ],
  )
```

Una vez hecho todo esto, se ha de ejecutar un comando que generará unos archivos en formato *.arb* para cada idioma con nombres en forma de: *intl_<idioma>.arb*. Estos archivos contendrán una lista con todos los mensajes, y el desarrollador debe modificar cada uno para cada uno de los idiomas a los que se está traduciendo.

Después ha de ejecutarse un segundo comando que importará las traducciones.

Finalmente, para introducir los mensajes en la lógica de la app, se realiza una llamada como la siguiente:

```
AppLocalizations.of(context).appTitle
```

5.5 Entorno de desarrollo

5.5.1. Control de versiones

Sistema por defecto

En las *Alexa-hosted* skills, se incluye un sistema de control de versiones por defecto, *AWS CodeCommit* (basada en *Git*). Este debe ser utilizado de forma obligatoria para desplegar la skill.

Este sistema está estructurado en tres ramas[16]:

- *master*: Código desplegado en fase de desarrollo.
- *prod*: Código desplegado en fase de producción (sólo se puede subir código a ella si la skill está en producción).
- *dev*: Código sin desplegar en fase de desarrollo, mostrado en el editor de código de la consola de desarrollador. Esta rama en realidad no se utiliza, pues pertenece a la antigua versión 1 del *ASK CLI*, y en teoría no tiene uso en la actual versión 2. Sin embargo, si esta rama no se encuentra a la par con *master*, el editor de código de la consola deja de funcionar.

Complejidad añadida

Sin embargo, esto no era suficiente para la skill de "Al Loro" debido al lenguaje de programación que utiliza: *TypeScript*. Las *Alexa-hosted* skills sólo soportan *JavaScript* y *Python*; por tanto, el código *TypeScript* debía ser compilado previamente a *JavaScript* para luego desplegar la skill con el código resultante. Esto implica que el único código con control de versiones es el código compilado, cuando lo que interesa es que sea el código fuente el que lo esté.

Para lograr este objetivo, se optó por utilizar *GitHub*[17] (plataforma también basada en *Git*) como sistema de control de versiones para el código fuente. Ahora bien, al tener dos

[16] https://developer.amazon.com/es-ES/docs/alexa/hosted-skills/alexa-hosted-skills-ask-cli.html

[17] https://github.com/

sistemas distintos dentro del mismo proyecto, era necesaria una estructura de proyecto más compleja para poder gestionarlo más fácilmente.

Primer diseño

El primer diseño para la estructura del proyecto fue el siguiente:

- Disponer de un único repositorio.
- Además de las ramas incluidas por Alexa, se crea una nueva rama de nombre "*codebase*" (base de código).
- Esta nueva rama tiene únicamente el código fuente, configurado para su sincronización con la plataforma *GitHub*.
- Se trabaja desde la rama *master*, y se utilizan *hooks* (segmentos de código que se ejecutan automáticamente cuando ocurren ciertos eventos dentro del flujo de trabajo) para aplicar los cambios de esta rama en *codebase*.

Tras llevarse a la práctica, este diseño resultó tener un importante **defecto**: utilizaba la funcionalidad de las ramas de una manera para la que no estaba concebida.[18]

En el sistema de control de versiones *Git*, el concepto de ramas está basado en la idea de que tienen un origen común, y aplicar cambios de una a la otra utiliza este origen como punto de referencia. Sin embargo, este modelo requiere de que el contenido de un archivo sea completamente distinto entre las dos versiones: "*.gitignore*".

Este archivo de configuración indica al sistema *Git* qué archivos o carpetas no deberían formar parte del control de versiones (a esto se le llama "**ignorar**"). En las ramas de Alexa, el código compilado *debe* formar parte del sistema para poder ser desplegado. En cambio, en la rama de GitHub no interesa, ya que el código fuente es más que suficiente.

Esto provoca un conflicto cada vez que se realizan cambios a este archivo en una rama de Alexa y se intentan aplicar a la rama de GitHub. Se debe resolver manualmente cada vez que ocurre, dando preferencia a la versión de GitHub para evitar que se apliquen, o editando el archivo para que se ajuste a la otra configuración, según el caso.

Por otra parte, este modelo es bastante poco flexible. Un ejemplo de ello se vería si se pretendiese crear una nueva rama para implementar una funcionalidad que no se desea desplegar aún, pero en la que se quiere trabajar localmente. Cuando llegara el momento de aplicar los nuevos cambios a las ramas de Alexa, se encontraría con una serie de inconveniencias que le obligarían a hacer más trabajo del necesario.

Modelo final

Tras un tiempo con el primer diseño, se ideó un nuevo modelo mejorado que resolvería los problemas anteriormente mencionados. La estructura es la que sigue:

- El código se separa en dos repositorios: Uno principal que se sincroniza con GitHub y contiene sólo la base del proyecto (el código fuente y el *skill package*); y otro secundario, que recibe el código compilado y una copia del *skill package*, y se encarga de hacer los despliegues en Alexa.

[18] En el momento de su ideación, este defecto era conocido. Sin embargo, la falta de resultados en la búsqueda de un modelo adecuado llevó a la implementación de este diseño de manera temporal.

- El repositorio principal se encuentra en la carpeta raíz del proyecto, y el secundario en una subcarpeta de nombre "*build*", ignorada en el primero.

- El repositorio principal tiene, a su vez, una rama principal (de nombre "*main*"). Cada vez que se realicen cambios sobre ella, se ejecutará un código que, en base al tipo de contenido que ha cambiado, podrá hacer lo siguiente para aplicar los mismos cambios al repositorio secundario: compilar el código, copiar el *skill package*, y/o copiar archivos de configuración necesarios (como puede ser el archivo *package.json*, que contiene, entre otras cosas, la lista de dependencias[19] del proyecto).

5.5.2. Editor y Depuración

Editor utilizado

El editor utilizado para desarrollar la skill ha sido *Visual Studio Code*[20], desarrollado por *Microsoft*. En un principio se trataba simplemente de una preferencia personal, pero más adelante Amazon desarrolló una extensión para dicho editor sobre el desarrollo de skills en Alexa, llamada *ASK Toolkit*[21], que añade una serie de funcionalidades muy útiles.

Depuración local con simulador

La principal funcionalidad de la extensión de Alexa es su **simulador de skills local**[22]. La consola de desarrollador ya contiene uno para poder hacer pruebas sobre la skill ya desplegada. No obstante, este simulador permite probar skills que aún no han sido desplegadas, pudiéndose así modificar el código más fácilmente.

Además, esta funcionalidad se complementa muy bien con una librería oficial disponible para hacer **depuración local**, llamada *ask-sdk-local-debug*[23].

5.6 Contratiempos

5.6.1. Límite de caracteres en una respuesta

En la documentación de Alexa acerca del formato de una respuesta de la skill[24], se comentan los límites de caracteres en varios campos. Por ejemplo, el texto con la respuesta que Alexa debe decir al usuario (*outputSpeech*) no debe superar 8000 caracteres. La misma restricción se aplica al texto de una *tarjeta* (cuadro mostrado en la app de Alexa).

Para los dos casos comentados anteriormente, se puede implementar con relativa facilidad cierta lógica que haga un truncado del texto y, si le falta texto por leer, pregunte al usuario si quiere que siga leyendo (de hecho, es precisamente esto lo que se hace en "Al Loro").

[19] Una dependencia es una librería de la que depende el proyecto para poder funcionar.
[20] https://code.visualstudio.com/
[21] https://developer.amazon.com/es-ES/docs/alexa/ask-toolkit/get-started-with-the-ask-toolkit-for-visual-studio-code.html
[22] https://developer.amazon.com/es-ES/docs/alexa/ask-toolkit/vs-code-testing-simulator.html
[23] https://github.com/alexa/alexa-skills-kit-sdk-for-nodejs/tree/2.0.x/ask-sdk-local-debug
[24] https://developer.amazon.com/es-ES/docs/alexa/custom-skills/request-and-response-json-reference.html#response-format

Sin embargo, existe un campo en cada respuesta que puede potencialmente tener una gran cantidad de datos: las *SessionAttributes* (4.2.4). En el caso de la skill desarrollada, esto resulta en un problema, pues se deben guardar ahí todos los elementos de la feed que se está leyendo para no tener que hacer peticiones innecesarias a la web que la aloje. Si la feed tiene muchos elementos, o estos son muy largos, acaba superando uno de estos límites y devolviendo un error (además, uno genérico que no da ninguna información al usuario sobre el motivo por el que ha ocurrido, ni su solución).

En la documentación no se especifica un límite concreto para el campo de las *SessionAttributes*. Sin embargo, sí se indica un límite general en el tamaño de la respuesta completa: debe ser, como máximo, de 24 kilobytes.

El primer paso para tratar de mitigar este problema fue añadir una opción para que el usuario pueda limitar el número de elementos de una feed. Esto ayudaba en algunas feeds que tenían un número elevado de elementos (aunque no era lo habitual). Sin embargo, esto no era suficiente: podría seguir ocurriendo, y el error seguiría sin dar información al usuario. Por tanto, lo ideal sería lograr saber exactamente cuándo se iba a producir este error y prevenirlo lanzando un error personalizado con más información.

La primera posible solución fue asumir que 24 KB equivalen a 24000 caracteres. En ese caso, se podría obtener la respuesta completa, contar sus caracteres, y dar un error si superan este número. Pero resulta que no se puede obtener la respuesta completa (o al menos no de forma sencilla), con lo cual se acabó descartando esta idea.

Tras esto, se propuso hacer algo similar a lo anterior, pero con las *SessionAttributes* (a las que sí se tiene acceso). De los 24000 caracteres asumidos en total, se pueden recortar unos 8000 para la respuesta oral, y otros 1000 como margen. Esto resultaría en 15000 caracteres. Igual que con la idea anterior, si el objeto como cadena de texto supera dicha cantidad de caracteres, se lanza un error.

Esto en teoría podría funcionar; no obstante, haciendo pruebas se descubrió que el asumido límite de 24000 caracteres no se aplicaba a la realidad. De hecho, era muy superior. Se comprobó que, con unas *SessionAttributes* de 65536 caracteres, seguía funcionando con normalidad. Se fue probando con otros valores, y con 99326 caracteres no funcionaba. Por tanto, el límite real debe encontrarse entre estos dos valores; pero se consideró que no valía la pena ser más exhaustivo con la búsqueda de este.

Finalmente, se encontró un valor arbitrario entre los dos que funcionaba con todas las feeds que se probaron, y da un error si la lista de elementos de una feed como cadena de texto supera dicha cantidad.

En el caso de que la skill hubiera acabado publicándose, probablemente se habrían hecho pruebas más exhaustivas y se habría tratado de buscar una mejor solución (posiblemente aplicando a cada feed un truncado automático de su contenido para evitar esta situación). Aun así, al tratarse de un caso extremo, no merecería una prioridad demasiado alta (a menos, claro, que empezara a darse en un mayor número de ocasiones).

5.6.2. Validación de skills y base de datos

Uno de los procesos automatizados necesarios para realizar la certificación de la skill (y así poder publicarla) es la **validación** (6.1.2).

Tras probar este sistema para ver cómo funcionaba, se encontraron múltiples nuevos usuarios en el sistema de autenticación y en la base de datos. Además, la base de datos había sufrido un número de consultas mucho más alto de lo habitual.

El motivo resultó ser que la validación implicaba iniciar la skill múltiples veces; pues durante el lanzamiento de la misma se consultaban las feeds del usuario, y se creaba uno nuevo en el caso de que no existiese.

Como resultaba un poco extraño que esto ocurriese a propósito (especialmente por el hecho de que no se indica en ningún sitio que esto vaya a pasar), se pensó que podría deberse a que el sistema de base de datos no hacía uso inicialmente de la interfaz *PersistenceAdapter* (4.2.4), la forma "oficial" de implementar la persistencia entre sesiones.

Esto llevó a un proceso de reorganización de código para que utilice este sistema. Las tres funcionalidades donde se interactuaba con una base de datos (obtener la lista de feeds, crear un nuevo usuario, y generar un código de autenticación) fueron adaptadas para que internamente actuaran en base a las funciones disponibles en la interfaz *PersistenceAdapter*. La clase donde solían residir dichas funcionalidades, que abstraía la persistencia de la lógica (de nombre "*Database*") continuó existiendo para mantener la abstracción y, además, no tener que modificar la lógica durante este proceso.

Una vez terminado se volvió a ejecutar la validación, y seguía ocurriendo lo mismo. Sin embargo, gracias a la necesaria revisión del código de persistencia, se lograron resolver y/o mejorar algunas cuestiones.

Por ejemplo, se modificó el código para que sólo se creen nuevos usuarios una vez pidan iniciar sesión (en lugar de hacerlo al iniciar la skill). Esto redujo significativamente el número de usuarios que la validación creaba (aunque no llegaron a cero).

Otra mejora fue limitar los códigos de inicio de sesión a uno por usuario (aunque esto no resolvió nada relacionado con el problema comentado).

No se logró resolver por completo el problema mencionado en esta sección (por motivos ajenos al desarrollo de la skill), pero sigue siendo una mejora.

5.6.3. Migración del sistema de autenticación

Originalmente, la API encargada de la validación de códigos en el sistema de autenticación (4.1.2) estaba hospedada en otro servicio: *Firebase Functions*. Sin embargo, un suceso y una situación hicieron necesaria una migración a otra plataforma:

- El suceso fue que, un tiempo después de ser publicada, la API sufrió un ataque de fuerza bruta (es decir, recibió una gran cantidad de llamadas con muy poco retardo entre ellas conteniendo códigos al azar). Esto dio a entender que era muy necesaria una protección adicional a la API, tratando de conseguir que sólo la app tenga acceso.
- La situación era que el servicio utilizado dejaba de formar parte del plan gratuito *Spark* y pasaba al plan *Blaze*, con formato "paga lo que necesites". Seguía teniendo una capa gratuita amplia (lo cual habría sido probablemente suficiente), pero obligaba a introducir datos de pago (en caso de que no lo fuese).

Se sopesaron varios servicios, valorando especialmente las opciones de protección que ofrece cada uno. Finalmente, se acabó utilizando la plataforma de Amazon *AWS*, ya que dispone de soporte para protección mediante *API keys*[25]. Como desventaja, también es obligatoria la introducción de datos de pago para utilizar los servicios necesarios en este caso; y además, uno de ellos sólo está disponible de forma gratuita durante los primeros 12 meses. A pesar de esto, se consideró que la protección incluida por defecto era más útil e importante.

[25] Las *API keys* son claves generadas automáticamente que se utilizan para autorizar peticiones.

CAPÍTULO 6
Implantación

6.1 Certificación de la skill

6.1.1. Datos de distribución

Para poder publicar la skill, se deben rellenar una serie de datos relacionados con su distribución. Entre ellos se encuentran:

- **Información** a mostrar en la **tienda de skills** para todos los idiomas en los que vaya a estar disponible. Ejemplos son: nombre, descripción, frases de ejemplo, icono, etc.
- Aspectos legales, centrados en la **privacidad** y la **conformidad** del desarrollador a ciertas acciones que desde Amazon puedan aplicar a tu skill.
- Selección de **disponibilidad** de la skill. Contiene opciones como quién debería tener acceso a la skill o las regiones donde el desarrollador quiera que esté disponible.

6.1.2. Validación

Como parte del proceso de certificación, existe un sistema de validación de skills. Este aparece en la consola de desarrollo como un apartado con un solo botón para ejecutarlo, y una vez terminado pueden pasar dos cosas:

- En el caso de que haya errores, aparecen una serie de mensajes indicando cuáles han sido y su posible causa.
- Si no hay errores, se muestra un único mensaje con el texto "*Zero errors found*" (cero errores encontrados).

La falta de *feedback* por parte del sistema acerca de las pruebas que realiza sobre la skill es bastante inconveniente, pues pueden resultar en una ráfaga de llamadas a APIs externas o bases de datos (5.6.2).

Además, apenas hay documentación al respecto: en el artículo sobre la certificación[1] ni se nombra, y en la única página encontrada sobre esto[2] no se comenta nada sobre qué hace exactamente la validación.

[1] https://developer.amazon.com/es-ES/docs/alexa/certify/certify-your-skill.html
[2] https://developer.amazon.com/es-ES/docs/alexa/smapi/skill-validation-api.html

6.1.3. Presentación / Envío

El paso final en la certificación de una skill es su **envío** (en inglés *submission*), donde se realizará una **revisión manual** de la skill (la cual incluye una prueba funcional) para decidir si se da o no el visto bueno para certificarla.

Para el desarrollador esto puede resultar una traba, ya que alarga cualquier desarrollo sobre la skill al tener que revisarse cualquier cambio a ésta.

Sin embargo, la revisión manual se trata, probablemente, de una **medida de seguridad** por parte de Amazon para evitar que skills con contenido malicioso lleguen a los usuarios de Alexa.

Un factor relevante es que las skills pueden ser recomendadas por Alexa de forma automática si detecta que aquello que desea hacer el usuario puede ser satisfecho por una skill. Y sólo con que el usuario diga "sí" tras recibir una recomendación, dicha skill es instalada y ejecutada. Este medio no proporciona información alguna acerca del origen de la skill que se está recomendando al usuario, cosa que hace a este último asumir que viene de una fuente fiable. Por tanto, es clave que Amazon realice revisiones exhaustivas para asegurar esto.

En el caso de Al Loro, no se ha llegado a realizar este envío; pues, al no tener intención de mantener la skill más allá del desarrollo asociado a este trabajo, se consideró en su momento que no merecía la pena pasar por el proceso de revisión.

6.2 Publicación de la app (Android)

Para poder probar la app en un entorno de producción, a la vez que facilitar la instalación de nuevas versiones en los dispositivos que se usaron para hacerlo, se propuso crear una cuenta de desarrollador en la *Google Play Console*[3] (la plataforma de desarrollo para la tienda de aplicaciones del sistema operativo *Android*).

Una vez creada la cuenta, se creó la aplicación desde el panel de control. Se rellenaron todos los campos obligatorios en los dos idiomas en los que está disponible (inglés - Estados Unidos y español - España), además de incluir un icono para la app.

Tras esto, se creó una pista de pruebas internas. A grandes rasgos, se trata de un espacio donde uno puede subir versiones de la app que estarán disponibles sólo para un grupo de hasta 100 usuarios que uno elija, que harán de *testers* (es decir, probarán la aplicación). En este caso, sólo hubo un tester: el propio desarrollador.

Finalmente, se publicó la primera versión de la app. La primera vez que se hace una publicación, desde Google deben hacer una revisión manual y aprobarla antes de estar disponible (incluso en pruebas internas). Tardan unos días; y una vez es aprobada, las siguientes versiones son publicadas casi al instante.

La publicación fue un éxito: como se puede ver en la figura 6.1, la app acabó siendo aprobada, y todo funcionó correctamente.

[3] https://play.google.com/console/

Figura 6.1: Pantalla de pruebas internas de la consola de desarrollador de Google Play Console.

CAPÍTULO 7
Pruebas

7.1 Pruebas unitarias

7.1.1. Librería utilizada

Existen varias librerías de pruebas unitarias para JavaScript (*mocha, chai, jasmine,* etc.); pero la que más destaca entre la comunidad de TypeScript es ***jest***, debido a su popularidad y su sencilla integración con este lenguaje de la mano de la librería ***ts-jest***. Esta ha sido la ruta elegida para implementar las pruebas unitarias en "Al Loro".

7.1.2. Simulación de objetos con *mocks*

Una de las técnicas más útiles y potentes relacionadas con las pruebas unitarias es la del ***mocking***. Esta consiste en crear funciones u objetos "falsos"; es decir, que en lugar de actuar como sus contrapartes del código original, tienen un comportamiento propio, definido en el contexto de las pruebas unitarias.

Librería utilizada

Para hacer mocks de objetos en las pruebas unitarias de la skill se ha utilizado la librería ***ts-mockito***[1].

La librería tiene tres funciones principales:

- **mock**: Crea un objeto de tipo "*Mocker*" en base a la clase o interfaz pasada como tipo parametrizado.

- **when**: Esta función sirve para indicar qué debe hacer el objeto cuando una función o propiedad de éste sea llamada.

  La precondición puede ser más refinada: por ejemplo, haciendo que ocurran cosas distintas según los valores de entrada de la función. Funciones como **anything** o **anyString** sirven para este propósito.

  Puede devolver directamente un valor predefinido (**thenReturn**), resolverlo de forma asíncrona encapsulándolo en un objeto "Promise"(**thenResolve**), o llamar a una función personalizada (**thenCall**).

[1] https://www.npmjs.com/package/ts-mockito

- **instance**: Una vez definido el comportamiento que se quiere del mock, este método se encarga de crear la instancia final del objeto, lista para ser utilizada como un objeto más del tipo original. Este pequeño inconveniente es habitual entre las librerías de *mocking*, aunque algunas no requieren llamar a una función como esta.

Para arreglar o mejorar esta librería, también se dispone de dos funciones implementadas manualmente:

- **fmock**: Es una versión de **mock** que por defecto devuelve una referencia al propio objeto, en lugar de devolver *undefined* como hace la librería original.

  Esta función es útil para objetos diseñados con la idea (propia de la programación funcional) de que cada función devuelva al propio objeto para poder encadenar unas funciones con otras.

  Ese es el caso para el objeto "*ResponseBuilder*" del ASK SDK, y la solución sin *fmock* pasaba por llamar a **when** una vez por cada función que fuera a ser llamada en las pruebas unitarias, indicando que devolviera el propio objeto.

  Esto se volvió claramente insostenible a largo plazo, ya que cada vez que se creaba una prueba unitaria que llamaba a nuevas funciones de este objeto, se debía revisar de nuevo la definición del mock y añadir las declaraciones necesarias. Además, en algunos casos puede no ser tan evidente (ya que las llamadas pueden ser internas en la librería) y llevar a errores inesperados.

  Es por eso que se creó la función *fmock* para simplificar esta tarea.

- **resolvableInstance**: Esta función nace de un *bug* conocido de la librería *ts-mockito*. La solución fue propuesta por varios usuarios en la página asociada a este problema en GitHub[2], pero hasta ahora no ha sido incluida en la librería. Por tanto, se ha acabado introduciendo manualmente una versión de esta en "Al Loro".

Ejemplo práctico

El mock más utilizado es el creado para el objeto "*HandlerInput*" del SDK de Alexa.

El principal inconveniente que tiene este objeto y los mocks resuelven es que contiene una gran cantidad de atributos que deben ser inicializados y muchos de ellos no hacen falta para las pruebas unitarias.

Hacer un mock simplifica las cosas, ya que todos los atributos están vacíos por defecto y por tanto se pueden inicializar sólo aquellos realmente necesarios.

```
1  // HandlerInputMocks.ts
2
3  // Librerías relacionadas con Alexa
4  import { AttributesManager, HandlerInput, ResponseBuilder } from 'ask-sdk-
   core';
5  import { RequestEnvelope, Response, ui } from 'ask-sdk-model';
6
7  // Librería para hacer mocks
8  import { anything, instance, mock, when } from 'ts-mockito';
9
10 // Librería de localización
11 import { initNewInstance, TFunction } from '../../../src/util/localization';
12
13 // Funciones creadas manualmente para arreglar o mejorar la librería para
   hacer mocks
```

[2] https://github.com/NagRock/ts-mockito/issues/191

```
14  import { fmock } from '../ts-mockito/FunctionalMocker';
15  import { resolvableInstance } from '../ts-mockito/resolvableInstance';
16
17  // ...
18
19  export async function mockHandlerInput(
20  // Los parámetros pasados permiten personalizar el estado del objeto,
    pudiendo así probar casos concretos
21  {
22      locale,
23      sessionAttributes = {},
24      requestAttributes = {},
25      outputSpeech,
26      addTFunctionToRequestAttributes = true,
27  }: MockHandlerInputOptions = {}): Promise<HandlerInputMocks> {
28      const mockedHandlerInput = mock<HandlerInput>();
29
30      const mockedAttributesManager = mock<AttributesManager>();
31
32      // Para la localización no es necesario un mock,
33      // ya que lo único que es útil personalizar es
34      // la existencia (o no) del texto locale
35      const t = locale
36          ? await initNewInstance(locale)
37          : locale === null
38              ? () => ''
39              : () => {
40                  throw new Error(
41                  "You called the 't' function but never initialized the locale"
42                  );
43              };
44
45      // ...
46
47      // Estas funciones ahora devuelven los objetos obtenidos como parámetros
48      // y son las más útiles para simular diferentes estados en la skill
49      when(mockedAttributesManager.getRequestAttributes()).thenReturn(
50          requestAttributes
51      );
52
53      when(
54          mockedAttributesManager.getSessionAttributes<{
55          [key: string]: any;
56          }>()
57      ).thenReturn(sessionAttributes);
58
59      // Llamar a esta función con cualquier valor ahora no hace nada,
60      // en lugar de lanzar un error
61      when(
62          mockedAttributesManager.setSessionAttributes(anything())
63      ).thenCall(() => {});
64
65      // Para cada objeto interno que sea necesario, se crean instancias de
    mocks...
66      const mockedRequestEnvelope = mock<RequestEnvelope>();
67      const instanceRequestEnvelope = instance(mockedRequestEnvelope);
68
69      // ...para luego devolverlos en las propiedades adecuadas
70      when(mockedHandlerInput.requestEnvelope).thenReturn(
    instanceRequestEnvelope);
71
72      const instanceAttributesManager = instance(mockedAttributesManager);
73
74      when(mockedHandlerInput.attributesManager).thenReturn(
```

```
75            instanceAttributesManager
76        );
77
78        const mockedResponseBuilder = fmock<ResponseBuilder>();
79
80        const mockedResponse = mock<Response>();
81
82        when(mockedResponse.outputSpeech).thenReturn(outputSpeech);
83
84        // Como el objeto Response se devuelve de forma asíncrona
85        // dentro de la función getResponse(),
86        // se utiliza esta función especial para crear una instancia compatible
87        const instanceResponse = resolvableInstance(mockedResponse);
88
89        when(mockedResponseBuilder.getResponse()).thenReturn(instanceResponse);
90
91        const instanceResponseBuilder = instance(mockedResponseBuilder);
92
93        when(mockedHandlerInput.responseBuilder).thenReturn(
    instanceResponseBuilder);
94
95        const instanceHandlerInput = instance(mockedHandlerInput);
96
97        // Una vez creados todos los mocks y entrelazados unos con otros,
98        // se devuelven todos dentro de un solo objeto
99        // para facilitar un mayor nivel de personalización
100       // en cada prueba unitaria
101       const mocks: HandlerInputMocks = {
102           mockedHandlerInput,
103           mockedRequestEnvelope,
104           mockedAttributesManager,
105           mockedResponseBuilder,
106           mockedResponse,
107
108           instanceHandlerInput,
109           instanceRequestEnvelope,
110           instanceAttributesManager,
111           instanceResponseBuilder,
112           instanceResponse,
113
114           t: t,
115       };
116
117       return mocks;
118   }
```

Simulación de librerías

Un caso muy común es el de querer simular librerías enteras. Esto es especialmente útil cuando la librería en cuestión exporta funciones que son llamadas directamente en el código fuente.

Cualquier librería que haga llamadas externas (por ejemplo, peticiones a páginas web o APIs) es un buen sujeto para esta práctica.

La librería ***jest*** provee esta funcionalidad mediante la función ***mock(...)***, a la que se le pasa como cadena de texto el nombre de la librería (del mismo modo que la nombrarías al importarla), y crea un mock automático de la librería entera.

Opcionalmente, se le puede pasar también una función "*factory*" para definir manualmente la estructura de la librería. Esta opción es menos habitual, pero es la mejor cuando una librería tiene funciones dentro de objetos, y queremos que esas funciones internas

sean accesibles en forma de mock (ya que por defecto el objeto sería convertido en un mock vacío, dejando la función interna inaccesible).

Para acceder a los mocks de los elementos de la librería, es tan simple como instanciarlos, del mismo modo que uno lo haría con el elemento original.

Sin embargo, en TypeScript esto tiene una pega: el lenguaje no será capaz de detectar el cambio de tipo derivado del mock, y le asignará el tipo del elemento original. Esto implica que al intentar llamar a funciones propias del mock, TypeScript considerará esto un error. Para solventar esto, la librería ***ts-jest*** tiene la función ***mocked(...)***[3]. Llamando a esta función con el elemento como argumento, el tipo se resuelve correctamente.

```
1  // Ejemplo 1: mock automático
2  jest.mock('ask-sdk-core');
3
4  // Ejemplo 2: mock con factory
5  jest.mock('firebase-admin', () => ({
6    initializeApp: jest.fn(),
7    firestore: jest.fn(),
8    credential: {
9      cert: jest.fn(),
10   },
11 }));
12
13 // Ejemplo 3: Llamada a funciones de librerías con mocks
14 const intentName = "IntentDePrueba"
15 mocked(getIntentName).mockReturnValue(intentName);
16
17 expect(mocked(initializeApp)).toHaveBeenCalled();
```

Espías

Otra posible circunstancia es cuando el desarrollador quiere saber si una función ha sido llamada o no, cuántas veces, con qué argumentos... pero sin alterar el comportamiento original de la función.

Para ello existen los espías, funciones que permiten obtener este tipo de información. Tanto *jest* como *ts-mockito* tienen implementaciones de esta funcionalidad, pero la más conveniente es la de *jest*, ya que tiene una mayor versatilidad.

Para poder entender el siguiente ejemplo, se requiere una explicación previa:

- **Contexto:** En este conjunto de pruebas, la función a probar se llama "***truncateAll***". Esta función aplica una segunda función "***truncate***" un número determinado de veces, y el resultado final varía en función de los resultados intermedios de la función interna.

  Aquí la utilidad del espía es que podemos obtener estos resultados intermedios, y así probar que el resultado final es consistente.

- **Tipos de resultados:** En la funcionalidad de espía de la librería *jest*, un resultado puede ser de tres tipos:

  - ***Return***: Un resultado ha sido devuelto correctamente, y su valor está disponible.
  - ***Throw***: La función ha lanzado una excepción; y por tanto, no ha devuelto nada.

[3] La función no se puede importar desde el módulo "*ts-jest*" directamente, sino que hay que llamarlo "*ts-jest/utils*".

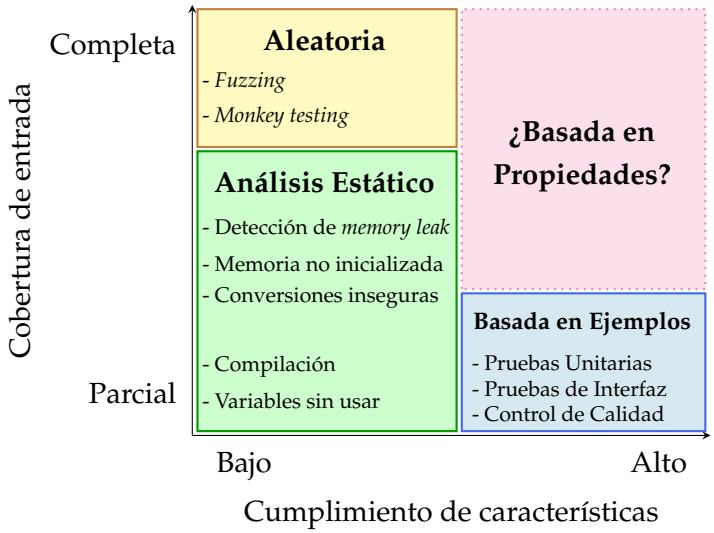
- ***Incomplete***: La función no ha terminado de ejecutarse, así que aún no ha devuelto ningún valor.

```
// truncateAll.test.ts

// ...

// Importando así la librería, se devuelve todo dentro de un objeto (truncate).
// Esto sirve para poder espiar sobre una función de la librería a través de este objeto
import * as truncate from '../../src/util/truncate';

// ...

describe('tests with truncate mocked', () => {
   // Espiamos la función, lo cual devuelve un objeto "espía"
   const truncateSpy = jest.spyOn(truncate, 'truncate');

   // Antes de cada prueba, se limpian los datos relacionados con las llamadas hechas a la función espiada
   beforeEach(() => truncateSpy.mockClear());

   // Esta función se utilizará en varias pruebas para obtener los resultados de cada llamada fácilmente
   function getTruncateSpyResults({ readable }: { readable?: boolean } = {}) {
       // Iteramos por la lista de resultados
       return truncateSpy.mock.results.map((res) => {
          // Si se ha devuelto algún valor, lo extraemos
          if (res.type === 'return') {
             return readable ? res.value.readable || res.value.str : res.value.str;
          } else {
             // Si no, falla la prueba indicando el error
             fail('One of the results was not returned, but instead: ' + res.type);
          }
       });
   }

   // Pruebas sobre truncateAll

   // ...

}
```

7.1.3. *Property based testing*

Las **pruebas basadas en propiedades** (*property based testing*) son un tipo especial de pruebas unitarias que logran una **mayor cobertura** de código, siendo capaces de **detectar errores** en **casos más aislados** y menos frecuentes (*edge cases*). Como se puede observar en la figura 7.1, las pruebas basadas en propiedades tienen el potencial de abarcar una nueva región que el resto de técnicas no llegan a cubrir del todo gracias a sus factores diferenciadores, los cuales se comentarán a continuación.

La técnica usada para diseñarlas tiene como mayor exponente la librería ***QuickCheck*** del lenguaje de programación **Haskell**, y consiste en **cambiar el enfoque** a la hora de crear pruebas unitarias. En lugar de buscar uno o varios ejemplos que abarquen los diferentes casos de uso de la funcionalidad a probar, **se indica** al propio **código** cómo puede



Eje	Significado
Cumplimiento de características	¿Prueba realmente la funcionalidad que intento implementar?
Cobertura de entrada	¿Cómo de bien cubro todos los posibles datos de entrada?

Figura 7.1: Técnicas de pruebas automatizadas disponibles (*Fuente:* [16])

crear sus propios ejemplos en base a una serie de **precondiciones** que todos ellos deben cumplir. Del mismo modo, el objetivo también cambia: cada prueba define una **propiedad** que la función debe cumplir. Estas propiedades tienen raíces matemáticas (producto de la naturaleza funcional[4] de Haskell), y suelen tener una estructura como la siguiente:

```
para todo (x, y, ...)

donde la precondición (x, y, ...) se cumple

la propiedad (x, y, ...) es verdadera
```

Ejemplo práctico

El siguiente ejemplo ha sido extraído de las pruebas unitarias de "Al Loro". La librería utilizada para implementar esta técnica se llama "*fast-check*", y está disponible para JavaScript y TypeScript a través de la plataforma *NPM*[5].

```
1  // Repeat.test.ts
2  import fc from 'fast-check';
3  // ...
4  test('works with ssml', async () => {
5   await fc.assert(
6     fc.asyncProperty(
7       // Crea cadenas de texto rodeadas por la etiqueta "ssml"
8       fc.string().map((str) => `<ssml>${str}</ssml>`),
9       async (ssmlOutput) => {
```

[4] https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional
[5] https://www.npmjs.com/package/fast-check

```
            // Crea un objeto de tipo SSML con el texto generado
            const ssmlOutputSpeech: ui.SsmlOutputSpeech = {
              type: 'SSML',
              ssml: ssmlOutput,
            };
            // Prueba la función de repetición con el objeto
            // y obtiene el resultado
            const lastResponse = await testSaveResponseForRepeatingInterceptor(
              ssmlOutputSpeech
            );

            // Comprueba que la respuesta guardada coincide con el ejemplo
            expect(lastResponse).toEqual(ssmlOutput);
          }
        )
      );
    });
```

Este ejemplo está basado en la funcionalidad de repetición. Esta consiste en que cada respuesta que devuelve Alexa es guardada para que si el usuario dice: "Repite", Alexa repita el mismo mensaje.

Cuando Alexa manda una respuesta hablada, puede hacerlo con dos tipos de objetos: SSML (*SsmlOutputSpeech*) o texto plano (*PlainTextOutputSpeech*). Esta prueba se asegura de que la función es capaz de manejar objetos del primer tipo.

7.2 Análisis estático

El **análisis estático** es una técnica basada en aplicar una serie de **reglas** al **código fuente** de una aplicación para identificar ciertos rasgos, normalmente asociados a la **calidad** y **seguridad** de dicho código.

La herramienta utilizada para ejecutar este tipo de análisis es ***Sonarqube***[6], una plataforma de código abierto (*open-source*) especializada en aplicar análisis estático para detectar posibles defectos en tres principales áreas:

- **Fiabilidad**: Este ámbito está compuesto únicamente de ***bugs***: segmentos de código considerados «defectuosos» al ser susceptibles de provocar errores o situaciones inesperadas.
- **Seguridad**
  - **Vulnerabilidades**: Se tratan de comportamientos conocidos por suponer un riesgo para la seguridad del software, y normalmente requieren acción inmediata por parte del desarrollador.

    Ejemplos de esto son el uso de sistemas de cifrado considerados inseguros, o la implementación de código susceptible de ser manipulado por agentes externos (como utilizar datos proporcionados por el usuario para realizar consultas a una base de datos).
  - ***Hotspots***: Este término, cuya traducción literal sería «puntos calientes», simboliza aquel código que requiere una revisión exhaustiva por parte del desarrollador para determinar si supone un riesgo de seguridad. Aunque en algunos contextos podría no serlo, su uso ha sido un origen común de vulnerabilidades conocidas.

[6] https://www.sonarqube.org/

- **Mantenibilidad**
  - ***Code Smells***: Esta expresión hace referencia a aquellos fragmentos de código que «huelen mal»; es decir, que tienden a incurrir en malas prácticas, y por tanto son más propensos a errores y problemas de mantenibilidad tales como la falta de legibilidad.
  - **Deuda técnica**: Es el tiempo estimado que le va a costar al desarrollador arreglar los problemas de mantenibilidad (principalmente los *code smells*).
    Cuanto mayor sea este valor, menor será la productividad en el proceso de desarrollo, ya que la cantidad de tiempo empleada en solventar problemas se podría haber destinado a implementar nuevas funcionalidades.

Además, tiene dos funcionalidades que Sonarqube no considera parte de la clasificación anterior:

- **Detección de código duplicado**: Muestra un porcentaje de líneas duplicadas. Por defecto, considera la prueba de calidad fallida si este valor supera el tres por ciento.
- **Cobertura de código**: Esta funcionalidad requiere una integración con la librería de pruebas unitarias (7.2.3), y muestra un informe de la cobertura de código por las pruebas existentes, indicando tanto un porcentaje general como una vista más detallada de cada fragmento de código.

7.2.1. Descripción de la herramienta

El sistema que conforma ***Sonarqube*** está compuesto por tres componentes (figura 7.2):

- **Cliente**
  - **Escáner**: Herramienta que analiza el código fuente y genera un informe con los resultados, el cual es enviado a través de HTTP al servidor principal.
- **Servidor**
  - **Servidor principal**: Es el encargado de procesar los resultados del informe, almacenarlos en la base de datos, y proporcionar al usuario con una interfaz y un sistema de búsqueda.
  - **Base de datos**: Servidor externo[7] donde se almacenan:
    - Las métricas y problemas generados por el escáner.
    - La configuración del servidor de Sonarqube.

7.2.2. Instalación y configuración

Servidor

Para instalar el servidor de ***Sonarqube***, existe una guía en la documentación oficial [17], la cual servirá de referencia en la siguiente descripción del proceso.

[7] En la versión 8.9, Sonarqube admite los siguientes motores de base de datos: PostgreSQL, Microsoft SQL Server y Oracle.

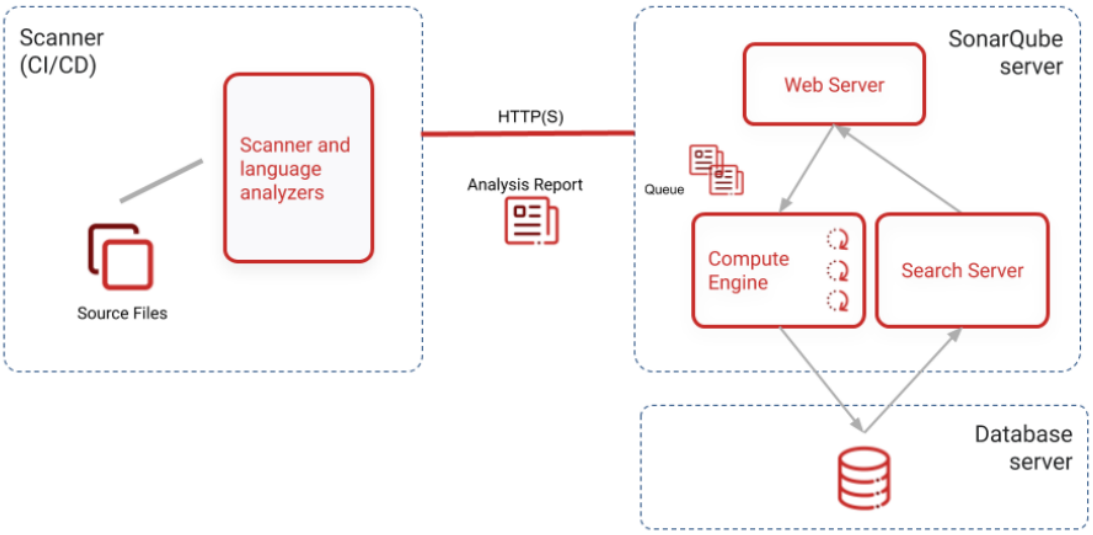


Figura 7.2: Arquitectura del sistema de Sonarqube (*Fuente:* [17])

Aunque el servidor se puede instalar de forma manual, el método que se va a discutir en esta sección es aquel que implica utilizar la plataforma ***Docker***[8] (concretamente la herramienta ***Docker Compose***[9]), ya que facilita mucho el proceso de instalación y mantenimiento.

En el caso de "Al Loro", se decidió instalar **localmente**, en lugar de utilizar un servidor dedicado. Esto se debe a que, aunque el procedimiento habitual es el último, se trata de un proceso más costoso, y cuya ventaja es solamente notable en proyectos con un ciclo de vida más largo y/o con un mayor número de desarrolladores.

Para instalarlo mediante *Docker Compose*, es necesario crear un archivo llamado "***docker-compose.yml***" con la configuración de los servicios que conforman el entorno del servidor (sonarqube + base de datos). Por suerte, en la documentación oficial se incluye una versión de ejemplo completamente funcional:

```
# docker-compose.yml

version: "3"

services:
  sonarqube:
    image: sonarqube:lts-community
    depends_on:
      - db
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
      - sonarqube_temp:/opt/sonarqube/temp
    ports:
      - "9000:9000"
  db:
    image: postgres:12
    environment:
```

[8] https://www.docker.com/
[9] https://docs.docker.com/compose/

```
24            POSTGRES_USER: sonar
25            POSTGRES_PASSWORD: sonar
26          volumes:
27            - postgresql:/var/lib/postgresql
28            - postgresql_data:/var/lib/postgresql/data
29
30      volumes:
31        sonarqube_data:
32        sonarqube_extensions:
33        sonarqube_logs:
34        sonarqube_temp:
35        postgresql:
36        postgresql_data:
```

Con este archivo y *Docker Compose* instalado, es tan sencillo como llamar al comando "***docker-compose up***" desde un terminal en la carpeta donde se encuentra el archivo. Este comando **instala** Sonarqube si no lo está y **lo inicia**.

La primera vez que se inicia, pedirá **usuario y contraseña**. Uno debe poner usuario "**admin**" y contraseña "**admin**". Una vez iniciada sesión, pedirá cambiar la contraseña. Deberás recordarla cada vez que inicies Sonarqube.

Para **parar el servidor** de Sonarqube, sólo hay que ejecutar el comando "***docker-compose down***" del mismo modo que se llamó al de inicio.

Para **desinstalarlo** completamente, se puede utilizar el comando "***docker-compose down -v --rmi 'all'***"[10] en su lugar, aunque es mejor **consultar la documentación**[11] **antes** de ejecutarlo.

Este sería el proceso habitual de instalación del servidor. Sin embargo, existe un pequeño **problema con los sistemas Linux**: existe un parámetro llamado "***vm.max_map_count***" que debe valer como mínimo "***524288***" para que Sonarqube funcione correctamente. Si no, lanza un error al iniciarse y no deja continuar.

Esto se puede arreglar **modificando este valor** de dos posibles formas (ambas requieren permisos de administrador):

- **Temporalmente**: Mediante el comando "*sysctl -w vm.max_map_count=524288*" se puede aplicar el cambio sólo para la sesión actual.
- **Permanentemente**: Se puede modificar el archivo de configuración "*/etc/sysctl.conf*" o crear uno en "*/etc/sysctl.d/99-sonarqube.conf*" para que utilicen este valor.

En el caso de una instalación **local**, lo más razonable es una opción **temporal**; ya que el dispositivo donde se instala tendrá más usos aparte de Sonarqube, y es probablemente mejor utilizar los valores por defecto en el resto de ocasiones.

En cambio, en una instalación en un **servidor dedicado** tendría más sentido la opción **permanente**, ya que ocurriría lo contrario que en la situación anterior.

Para facilitar el proceso de llamar a un comando más cada vez que se vaya a iniciar el servidor (para la solución temporal), se diseñó este *script*:

```
1   # up.sh
2
3   # Indica el programa con el que se debe ejecutar este script
4   # (esta debe ser la primera línea del script)
```

[10] Ten en cuenta que el comando anterior eliminará tanto la imagen de Sonarqube como la de la base de datos, lo cual podría dar problemas en otros proyectos con Docker. Para evitar esto, elimina el parámetro «*--rmi 'all'*» del comando, y desinstala la imagen de Sonarqube manualmente.

[11] https://docs.docker.com/compose/reference/down/

```
#!/bin/sh

# Versiones 8.4 en adelante
sonarqube_vm_max_map_count=524288
# Versiones 8.3 o anteriores
#sonarqube_vm_max_map_count=262144

# Obtén el valor actual del parámetro
current_vm_max_map_count="$(sysctl vm.max_map_count --values)"

# Si el valor actual es menor al valor mínimo que requiere Sonarqube,
# cámbialo por dicho mínimo
[ $current_vm_max_map_count -lt $sonarqube_vm_max_map_count ] && sudo sysctl
  -w vm.max_map_count=$sonarqube_vm_max_map_count

# Si el usuario actual es un miembro del grupo 'docker', no hacen falta
 permisos de administrador (obtenidos mediante el comando 'sudo')
if id -Gn | grep -q '\bdocker\b'
then
  docker-compose up -d
else
  sudo docker-compose up -d
fi
```

Cliente

Para instalar el escáner de Sonarqube (llamado ***SonarScanner***) de forma local con soporte para JavaScript/TypeScript, es necesario instalar una herramienta de **CLI**. Esta se encuentra disponible en forma de archivos **binarios** para Linux, Windows o Mac (64 bits), como una **imagen Docker**, o como un **binario especial** que requiere tener pre-instalada la **JVM** (Java Virtual Machine).

Sin embargo, existe una librería en la plataforma **NPM** llamada "***sonarqube-scanner***" (aunque no parece ser oficial) que encapsula el proceso de instalación y ejecución del escáner para poder hacerlo todo dentro del entorno de JavaScript. Se crea un archivo de este lenguaje llamando a la librería con la configuración adecuada, y así se puede ejecutar el escáner ejecutando el comando "***node analyse.js***" (o con el nombre de archivo que uno quiera):

```
// analyse.js

const fs = require('fs');
const scanner = require('sonarqube-scanner');

// Obtiene el token de un archivo
// (para evitar ponerlo en el propio código
//  y así poder incluir este archivo en el sistema de control de versiones)
const token = fs.readFileSync('./sonarqube/token.txt', 'utf-8').trim();

scanner(
  {
    token: token,
    options: {
      'sonar.login': token,
      'sonar.projectName': 'Feed the Parrot',
      'sonar.sources': 'src/',
      'sonar.tests': 'tests/',
      'sonar.javascript.lcov.reportPaths': 'sonarqube/coverage/lcov.info',
      'sonar.testExecutionReportPaths': 'sonarqube/test-report.xml',
    },
  },
```

```
23      () => process.exit()
24    );
```

7.2.3. Integración con las pruebas unitarias

Para poder mostrar datos relacionados con la cobertura de las pruebas unitarias en el informe de Sonarqube, es necesaria una integración con la librería que las ejecuta.

En este caso, para integrar ***jest*** con Sonarqube, se utiliza una librería llamada ***jest-sonar-reporter***. Esta es mencionada en la documentación de Sonarqube, y permite procesar los resultados del análisis de *jest* en un formato que el escáner de Sonarqube comprenda.

Para ello, una vez instalada la librería de integración, es necesario **configurar** *jest* para que haga uso de ella. Esto se puede lograr creando un archivo de configuración llamado "***jest.config.js***" en la carpeta raíz del proyecto:

```
1   // jest.config.js
2
3   const { defaults } = require('jest-config');
4   module.exports = {
5     preset: 'ts-jest',
6     testEnvironment: 'node',
7     testResultsProcessor: 'jest-sonar-reporter',
8     coverageDirectory: 'sonarqube/coverage',
9     coveragePathIgnorePatterns: [
10      ...defaults.coveragePathIgnorePatterns,
11      '<rootDir>/tests/helpers/',
12    ],
13  };
```

De este modo, cada vez que se ejecute el comando "***jest --coverage***" para analizar la cobertura de las pruebas, se creará un informe compatible con Sonarqube en la carpeta definida como "*coverageDirectory*".

CAPÍTULO 8
Conclusiones

8.1 Objetivos cumplidos

A lo largo del contenido de este trabajo, se ha hecho un **repaso** del **proceso de desarrollo** de la skill «Al Loro».

Para ello, se ha detallado el proceso de integración de una skill con servicios de **base de datos** tanto incluidos por Amazon (*DynamoDB*) como externos (*Firebase*).

Además, se ha implementado una **app** con *Flutter*, se ha seguido su proceso de implantación en Android, y se ha creado un método de **comunicación entre la skill y la app** a través de una *API* (para el servicio de autenticación) y de una base de datos en *Firebase*.

Por otro lado, se ha documentado el funcionamiento de una **interfaz** guiada por voz, presentando **ejemplos** en base al desarrollo de una **skill real** («Al Loro»).

También se han comentado los **servicios de *AWS*** que se ofrecen desde Alexa, y se ha profundizado en la **lógica de la skill** enseñando el modo de uso del ***ASK SDK***, el cual presenta grandes similitudes con librerías y herramientas típicas del **desarrollo *backend***.

8.2 Relación del trabajo desarrollado con los estudios cursados

Varios conceptos aprendidos en las asignaturas **AVD** (Análisis, validación y depuración de software) y **MES** (Mantenimiento y evolución de software) han sido utilizados para implementar las **pruebas** unitarias:

- Simulación de objetos con *mocks*
- *Property based testing*
- Análisis estático (*Sonarqube*)

Asimismo, los conocimientos acerca de la gestión de proyectos de asignaturas como **PSW** (Proceso de software) o **PIN** (Proyecto de ingeniería de software) han sido útiles para administrar un tablero kanban en *Trello*. Aunque la mejora habría sido más notable en un trabajo en grupo, ideas como la clasificación de tareas según su estado o el etiquetado en función de su prioridad y dificultad ayudaron a organizar mejor el proyecto y aumentar la productividad.

8.3 Aprendizaje, retos y dificultades

El principal desafío que ha supuesto el desarrollo de la skill ha sido la adaptación al entorno de implementar y mantener un software en producción, ya que surgen problemas que no se suelen tener en cuenta en un proyecto de carrera.

Por ejemplo, una decisión importante es la de elegir una plataforma donde alojar un servicio web. Para ello, no sólo hay que tener en cuenta el precio, sino también cuestiones de seguridad, funcionalidad; o, en proyectos más grandes, la capacidad de escalar (es decir, de poder lidiar con una carga de trabajo mayor, normalmente por tener un mayor número de usuarios).

Cuando se elige una plataforma menos adecuada para el tipo de servicio que uno busca ofrecer, pueden surgir una serie de problemas nuevos. Este fue el caso del servicio de autenticación, donde al elegir *Firebase Functions*, la *API* recibió un ataque de peticiones en masa (probablemente con el objetivo de obtener acceso a cuentas de usuario). Migrando el servicio a *AWS* se solucionó el problema, pero resultó en un trabajo extra que se podría haber evitado.

Otra lección obtenida gracias a este trabajo ha sido la de saber cuándo vale la pena invertir tiempo en arreglar un problema y cuándo no (especialmente cuando no tiene fácil solución).

Esto fue resultado de una limitación de Alexa: tiene un número máximo de caracteres en una respuesta, y cuando la noticia era demasiado larga, daba un error que el usuario no podía entender y se cerraba. En la documentación indica un número exacto de caracteres, pero en la práctica esto no se cumplía. Por tanto, la idea inicial fue probar distintas longitudes para tratar de dar con un número mágico a partir del cual fallara, y así poder comprobar con relativa facilidad esta condición; si se daba, se podría avisar al usuario.

Sin embargo, esta tarea resultó ser demasiado compleja. En cuanto se dio con un número lo suficientemente bueno como para que la probabilidad de error fuese bastante baja, se consideró que no merecía la pena continuar buscando una mejor solución.

Finalmente, el mayor aprendizaje ha sido el de aprender una nueva tecnología desde cero, documentar el proceso y tratar de crear un puente entre los procesos existentes y estas nuevas prácticas. Espero que el contenido de este trabajo pueda ayudar a desarrolladores que busquen iniciarse en este nuevo campo, y que logren implementar aquello que se propongan.

Bibliografía

[1] Documentación disponible sobre el asistente de Google.
https://developers.google.com/assistant/docs

[2] Documentación disponible sobre las Alexa skills.
https://developer.amazon.com/es-ES/docs/alexa/ask-overviews/what-is-the-alexa-skills-kit.html

[3] Documentación disponible sobre el kit de desarrollo para Siri (SiriKit).
https://developer.apple.com/documentation/sirikit

[4] Estudios sobre los diferentes asistentes de voz y sus sistemas de búsqueda.
https://www.semrush.com/blog/voice-search-local-seo/

[5] Informe realizado por *voicebot.ai* en 2020 sobre la cuota relativa de mercado de los asistentes de voz en *smartphones* de Estados Unidos.
https://voicebot.ai/2020/11/05/voice-assistant-use-on-smartphones-rise-siri-maintains-top-spot-for-total-users-in-the-u-s/

[6] Documentación acerca de las Alexa-hosted skills.
https://developer.amazon.com/es-ES/docs/alexa/hosted-skills/build-a-skill-end-to-end-using-an-alexa-hosted-skill.html

[7] Alexa Developer - Guía de migración del ASK CLI de la versión 1 a la 2.
https://developer.amazon.com/es-ES/docs/alexa/smapi/ask-cli-v1-to-v2-migration-guide.html

[8] Artículo en el blog de Alexa donde se comenta la existencia de la funcionalidad *"hook scripts"* a partir de la versión 1.5 del ASK CLI.
https://developer.amazon.com/es/blogs/alexa/post/89782c5d-6d3f-488c-843b-751656afc3c8/python-skill-development-now-supported-in-the-ask-cli-and-ask-toolkit-for-visual-studio-code

[9] Documentación acerca de la funcionalidad *"hooks"* en la herramienta de control de versiones *Git*.

- En inglés: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
- En español: https://git-scm.com/book/es/v2/Personalizaci%C3%B3n-de-Git-Puntos-de-enganche-en-Git

[10] Documentación sobre las *flash briefing* skills (skills que siguen el modelo de noticias)
https://developer.amazon.com/es-ES/docs/alexa/flashbriefing/understand-the-flash-briefing-skill-api.html

[11] Directrices relacionadas con la política de Alexa.
https://developer.amazon.com/es-ES/docs/alexa/flashbriefing/flash-briefing-skill-certification-checklist.html#policyGuidelines

[12] Lista de requisitos para completar con éxito el proceso de certificación de una skill de noticias.
https://developer.amazon.com/es-ES/docs/alexa/flashbriefing/flash-briefing-skill-certification-checklist.html

[13] Tabla con los elementos necesarios para una feed en una skill de noticias, y su formato adecuado.
https://developer.amazon.com/es-ES/docs/alexa/flashbriefing/flash-briefing-skill-api-feed-reference.html#details

[14] Requisitos del contenido textual de una feed en una skill de noticias.
https://developer.amazon.com/es-ES/docs/alexa/flashbriefing/flash-briefing-skill-api-feed-reference.html#text-content-requirements

[15] Documentación acerca del *APL* (*Alexa Presentation Language*).
https://developer.amazon.com/es-ES/docs/alexa/alexa-presentation-language/understand-apl.html

[16] Artículo: *Introduction to Property Based Testing* (*Medium*).
https://medium.com/criteo-engineering/introduction-to-property-based-testing-f5236229d237

[17] Sonarqube - Documentación oficial (versión 8.9 LTS).
https://docs.sonarqube.org/8.9/

APÉNDICE A
Manual de usuario (Skill)

A.1 Instalación

A.1.1. Como usuario

Alexa instala automáticamente las skills cuando el usuario pide abrirlas por primera vez. Por tanto, instalar esta skill es tan sencillo como decirle a Alexa: «Abre Al Loro»[1] (figura A.1).

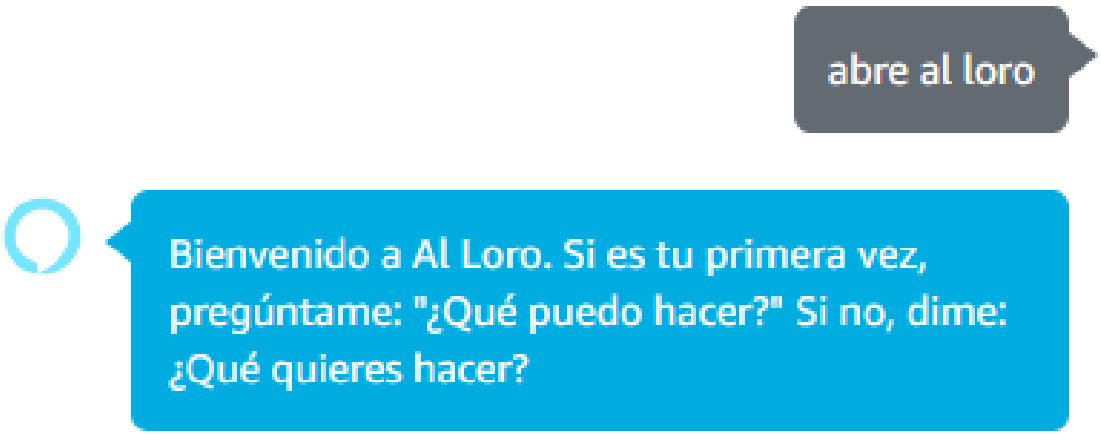
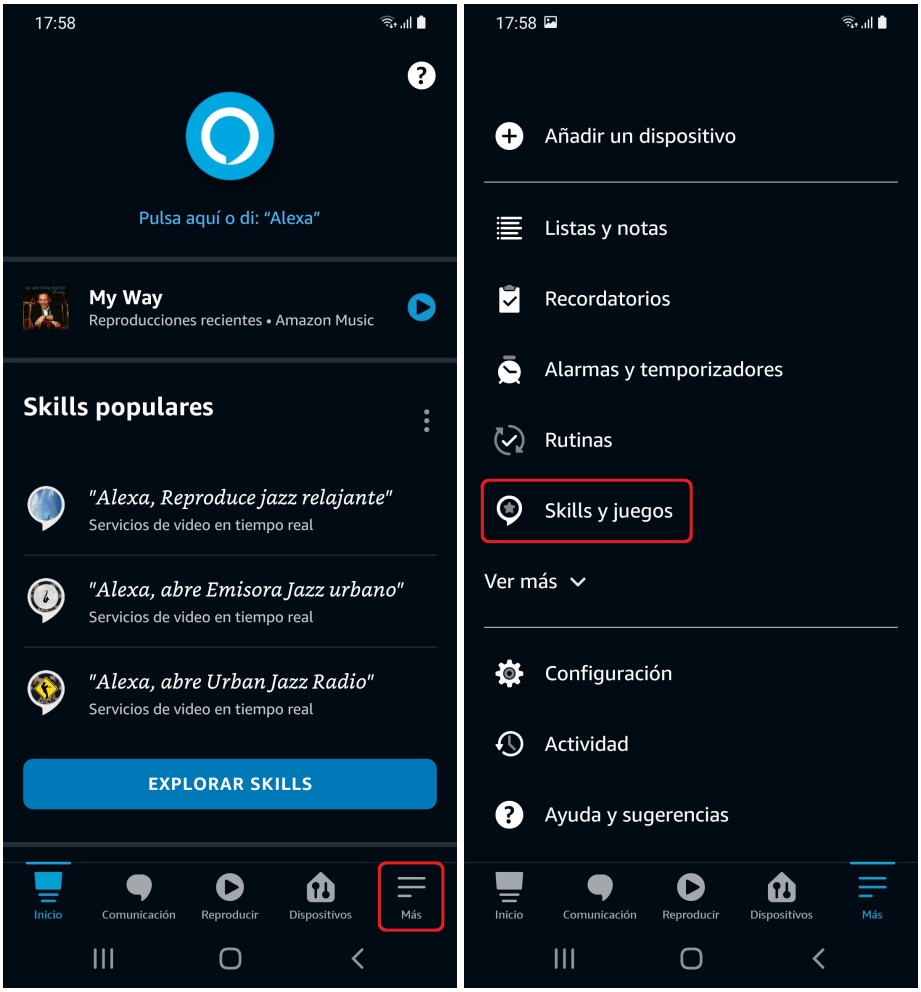


Figura A.1: Comando: Abrir la skill

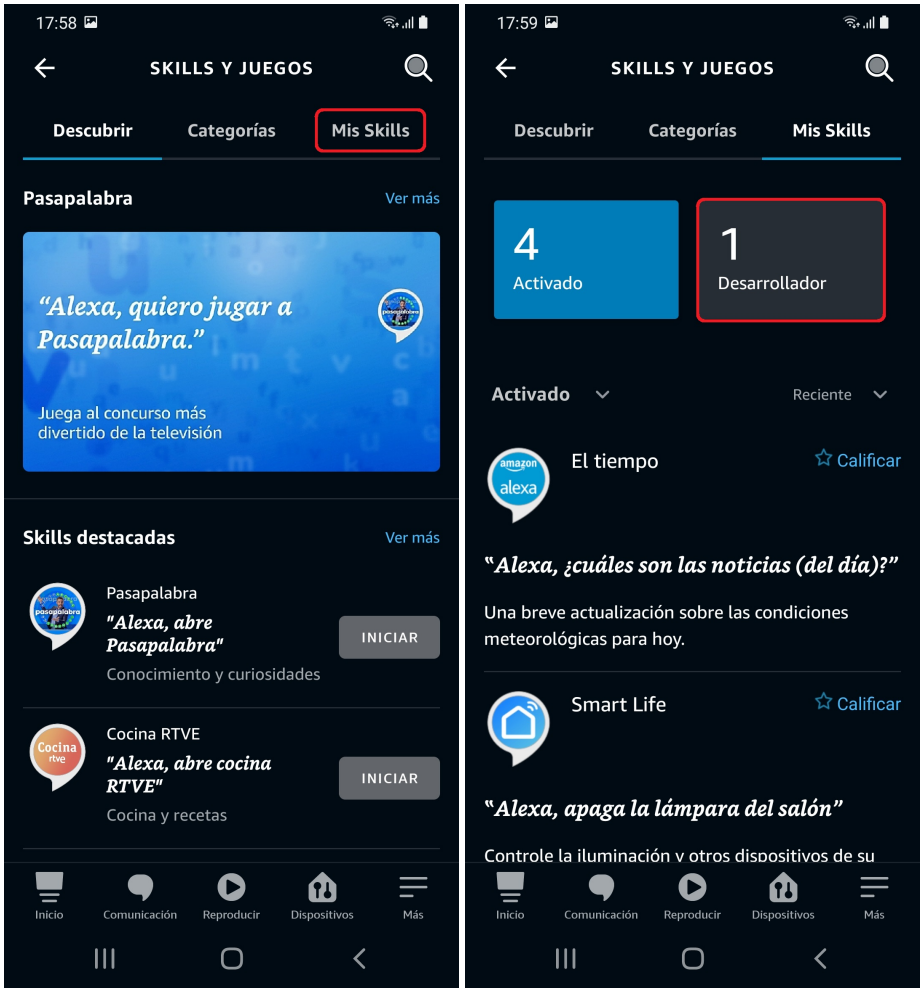
A.1.2. Como desarrollador

Para activar la skill como desarrollador, se debe utilizar la **app de Alexa**. Los pasos son:

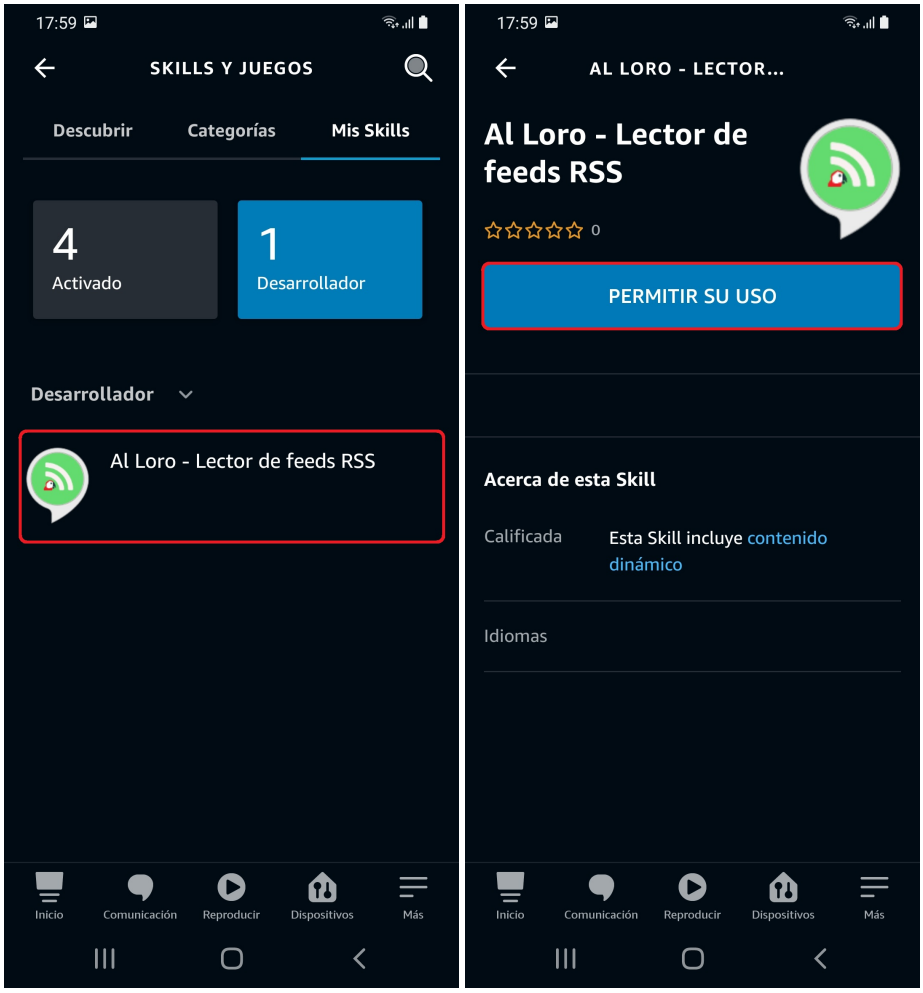
[1] Esto solo funcionaría si la skill hubiera sido aprobada tras un proceso de certificación y se encontrara en producción. En el caso de «Al Loro» no lo está, pues la certificación es compleja, requiere comunicación con el equipo de Amazon, y se consideró que no merecía la pena ya que no hay intención de mantenerla más allá de la duración de este trabajo.



1. Pulsar el botón «**Más**», situado en la esquina inferior derecha.
2. Acceder a la sección «**Skills y juegos**».



3. En la barra superior, pulsar «**Mis Skills**».
4. Pulsar sobre el cuadro «**Desarrollador**».



5. Si todo va bien, aparecerá la skill en una lista. Pulsar sobre ella.

6. Pulsar el botón «**Permitir su uso**».

A.1.3. Como beta tester

Existe una opción de *beta testing*[2] para el desarrollador, donde puede invitar a un máximo de 500 testers durante un periodo de hasta 90 días para que prueben la skill.

En este caso, es el desarrollador el que debe incluir a los testers añadiendo su dirección de correo electrónico a una lista. Una vez habilitado el beta test, se mandarán automáticamente correos a estas direcciones con un enlace a través del cual pueden instalar la skill tras iniciar sesión.

A.2 Comandos disponibles

- **Lee una feed**: También se puede llamar como «lee la feed {*nombre de la feed*}» para indicarla directamente (figura A.7). Si no se indica, Alexa preguntará por su nombre (figura A.2).

  Para ver el árbol de decisión tras invocar este comando, se puede observar el diagrama de flujo para este caso de uso de la figura 3.2.

[2] https://developer.amazon.com/en-US/docs/alexa/custom-skills/skills-beta-testing-for-alexa-skills.html

También existen mensajes de error personalizados para cuando una feed es inválida (figura A.5) o es demasiado larga (figura A.6).

- **Lista de feeds** (figura A.3)
- **Iniciar sesión** (figura A.4): Genera un código de seis dígitos para iniciar sesión desde la app. Tiene un periodo de validez de diez minutos; pasado ese tiempo, se debe generar un nuevo código.
- **Ayuda**: Lee una descripción general de los comandos disponibles.

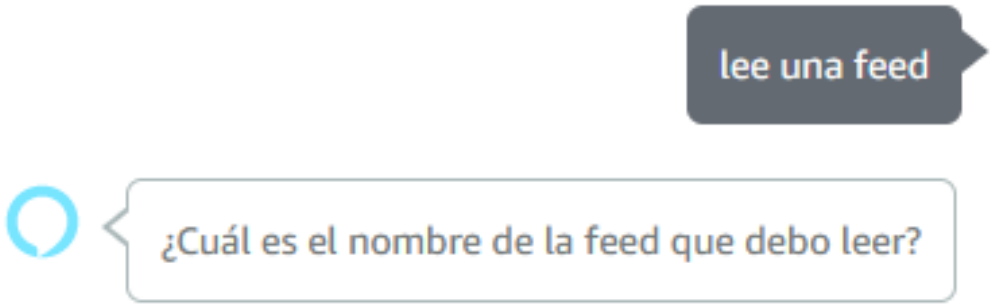


Figura A.2: Comando: Leer una feed (sin indicar el nombre)

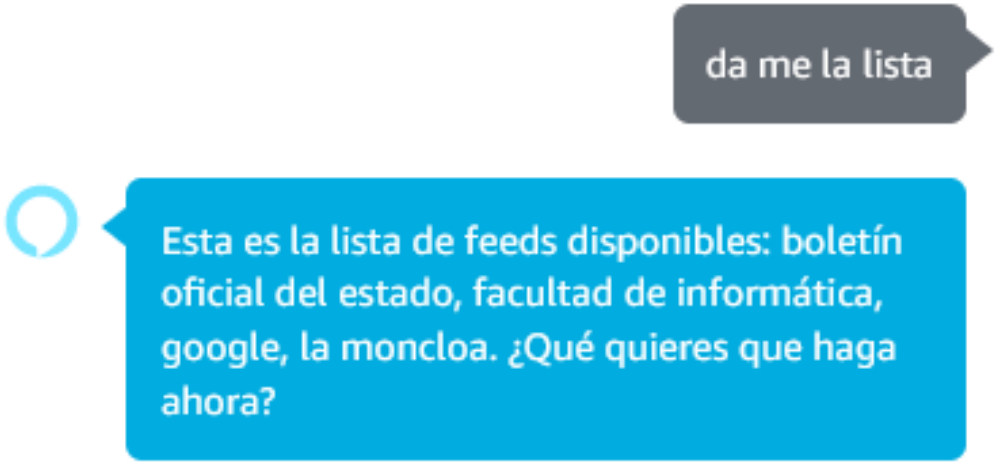


Figura A.3: Comando: Lista de feeds

A.3 Lista completa de comandos

En primer lugar, es importante mencionar que existen una serie de comandos que vienen predefinidos por Amazon. En estos casos, la lista completa no está disponible. Ejemplos son: «cancelar», «parar», «sí», «no», etc.

Los comandos predefinidos también pueden ser expandidos con mensajes personalizados que los activen. El ejemplo más claro en esta skill es el del comando «ayuda».

A continuación se muestra la lista completa de comandos personalizados con todas las posibles formas de invocar cada uno:

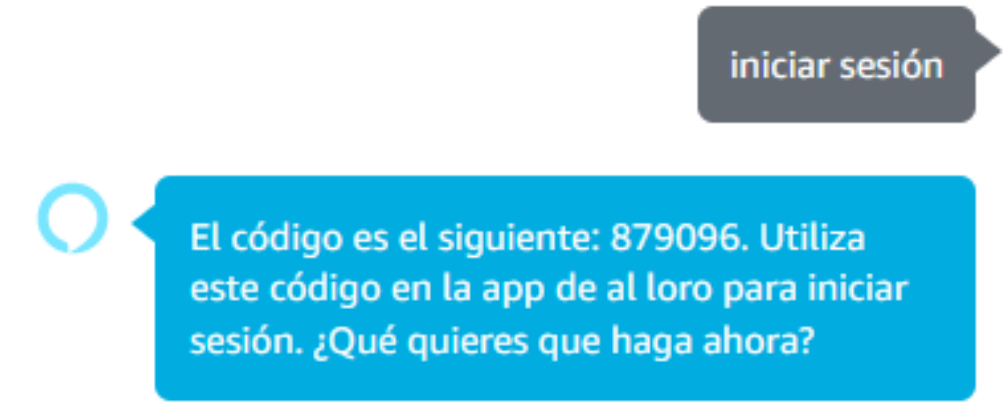


Figura A.4: Comando: Iniciar sesión

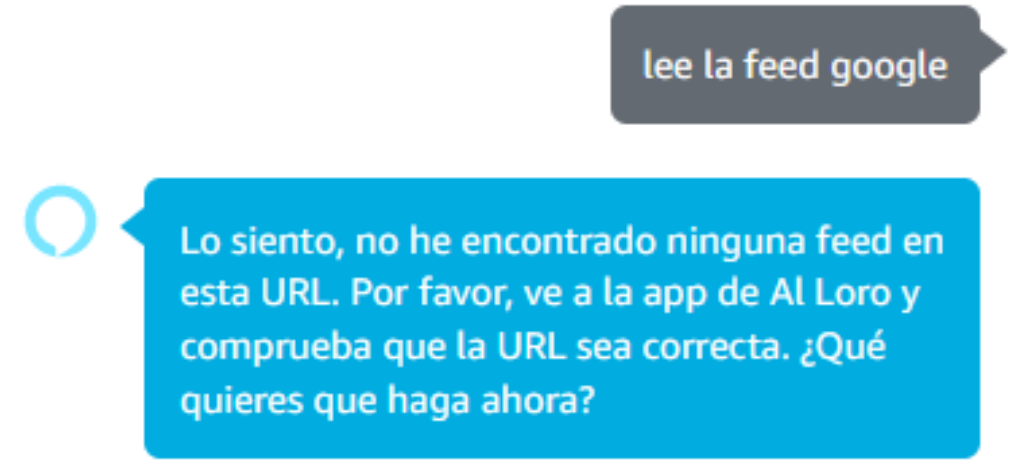


Figura A.5: Error: Feed inválida

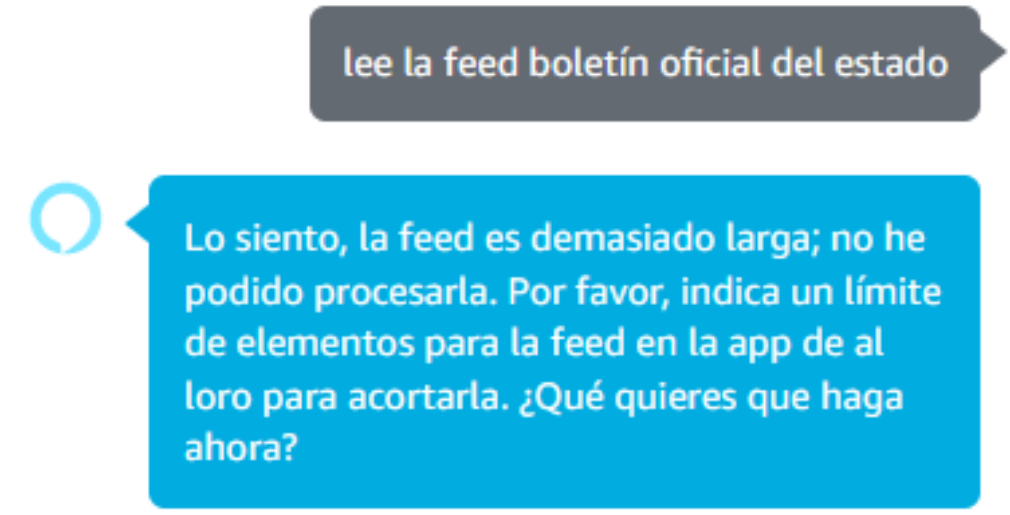


Figura A.6: Error: Feed demasiado larga

- **Leer una feed**:
  - «di las noticias de la feed {feed}»
  - «di las noticias»
  - «que diga las noticias de la feed {feed}»
  - «que diga las noticias»
  - «que me dé las noticias de la feed {feed}»
  - «que me dé las noticias»
  - «dame las noticias de la feed {feed}»
  - «dame las noticias»
  - «que me diga las noticias de la feed {feed}»
  - «que me diga las noticias»
  - «dime las noticias de la feed {feed}»
  - «dime las noticias»
  - «que me dé la feed {feed}»
  - «dime qué hay en la feed {feed}»
  - «dime qué hay»
  - «que me diga qué hay en la feed {feed}»
  - «que me diga qué hay»
  - «que me lea la feed {feed}»
  - «que me lea una feed»
  - «que lea la feed {feed}»
  - «que lea una feed»
  - «qué hay en la feed {feed}»
  - «dame la feed {feed}»
  - «léeme la feed {feed}»
  - «lee la feed {feed}»
  - «léeme una feed»
  - «lee una feed»
- **Leer una feed - Indicar nombre**:
  - «Se trata de la feed {feed}»
  - «Se trata de {feed}»
  - «Es {feed}»
  - «Es la feed {feed}»
  - «La feed se llama {feed}»
  - «Se llama {feed}»
  - «{feed}»
- **Lista de feeds**:
  - «que me diga qué feeds tiene disponibles»
  - «que me diga qué feeds tiene»
  - «que me diga qué feeds tengo»

  - «que me diga qué feeds tengo disponibles»
  - «que me diga la lista de feeds»
  - «que me dé la lista de feeds»
  - «lista de feeds»
  - «lista»
  - «dame la lista»
  - «dime la lista»
  - «dime la lista de feeds»
  - «dame la lista de feeds»
  - «qué feeds tienes disponibles»
  - «qué feeds tienes»
  - «qué feeds hay disponibles»
  - «qué feeds hay»
  - «qué feeds tengo disponibles»
  - «qué feeds tengo»
- **Iniciar sesión**:
  - «login»
  - «iniciar sesión»
  - «código para iniciar sesión»
  - «dame el código para iniciar sesión»
- **Ayuda** *(Predefinido por Amazon)*:
  - «qué puedo hacer»
  - «cómo puedo gestionar mis feeds»
  - «cómo gestionar mis feeds»
  - «cómo inicio sesión desde la app»
  - «cómo iniciar sesión desde la app»

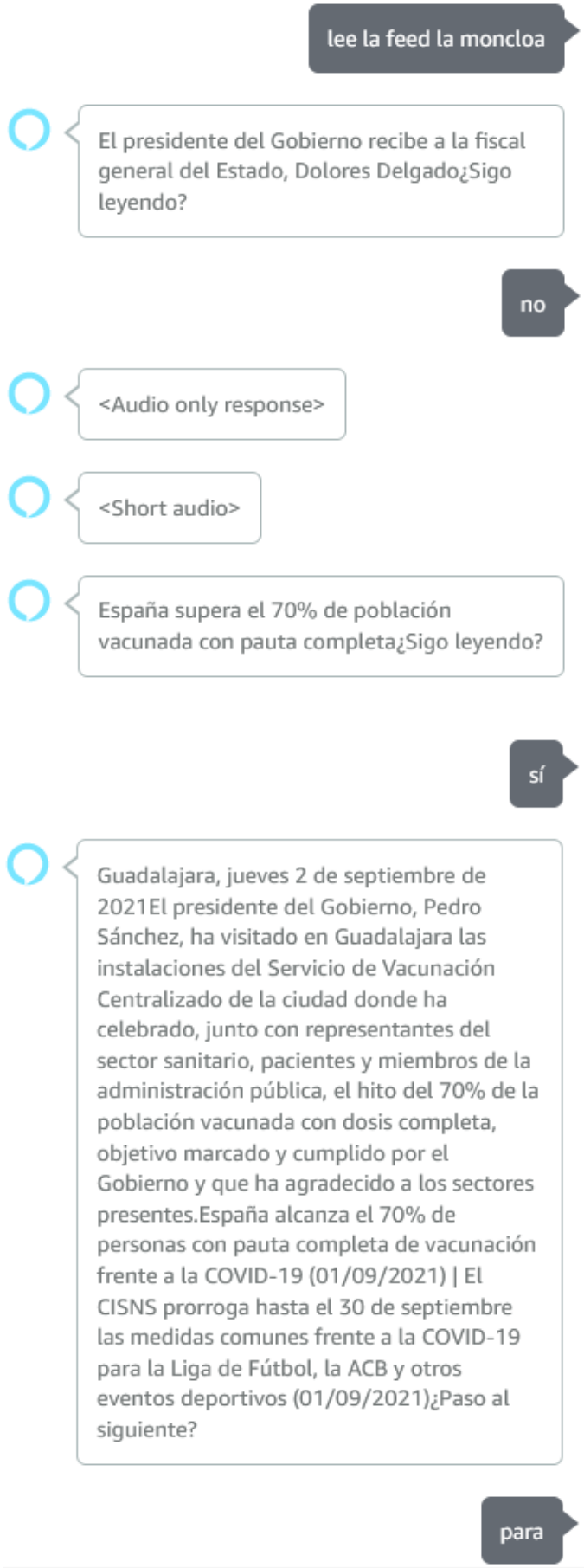


Figura A.7: Comando: Leer una feed

APÉNDICE B
Manual de usuario (App)

La app está formada por las siguientes pantallas:

- **Inicio sesión** (figura B.1): En esta pantalla se debe introducir un código que la skill dictó previamente al invocar el comando «iniciar sesión».
- **Gestión de feeds** (figura B.2): Muestra una lista de las feeds disponibles. Pueden seleccionarse con los recuadros de la izquierda para luego ser eliminadas con el botón de abajo, se pueden editar pulsando sobre el icono del lápiz, y se pueden crear nuevas pulsando sobre el botón «Nueva» (también situado abajo).

  Para cerrar sesión, se debe pulsar el botón situado en la esquina superior derecha.
- **Nueva / Editar feed**: Además de los campos básicos como el nombre, el idioma, o la URL (figura B.3), también tiene una serie de opciones para limitar el tamaño de la feed o para filtrarla por texto y/o categorías (figura B.4).

Figura B.1: App: Iniciar sesión

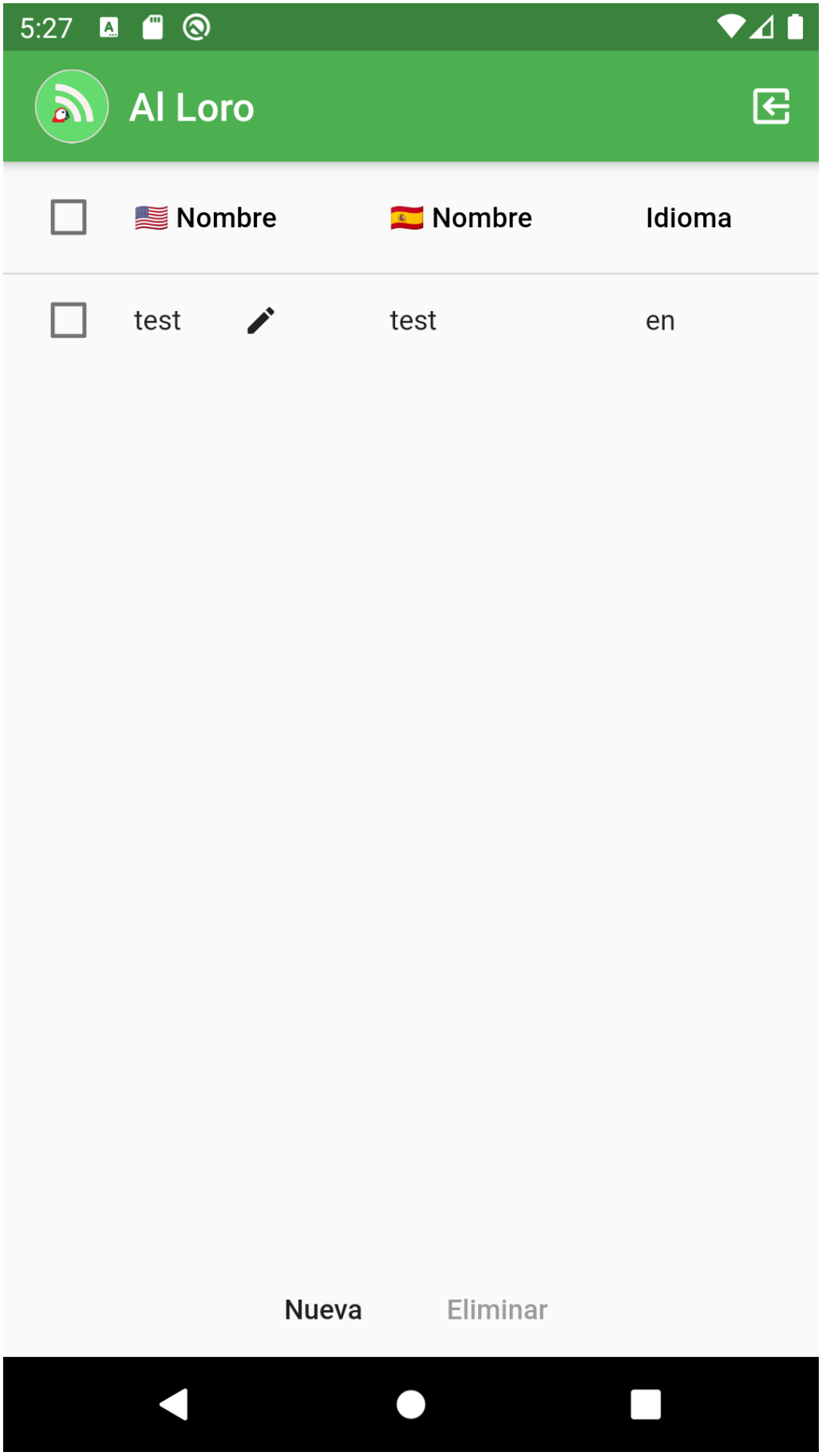


Figura B.2: App: Gestionar feeds

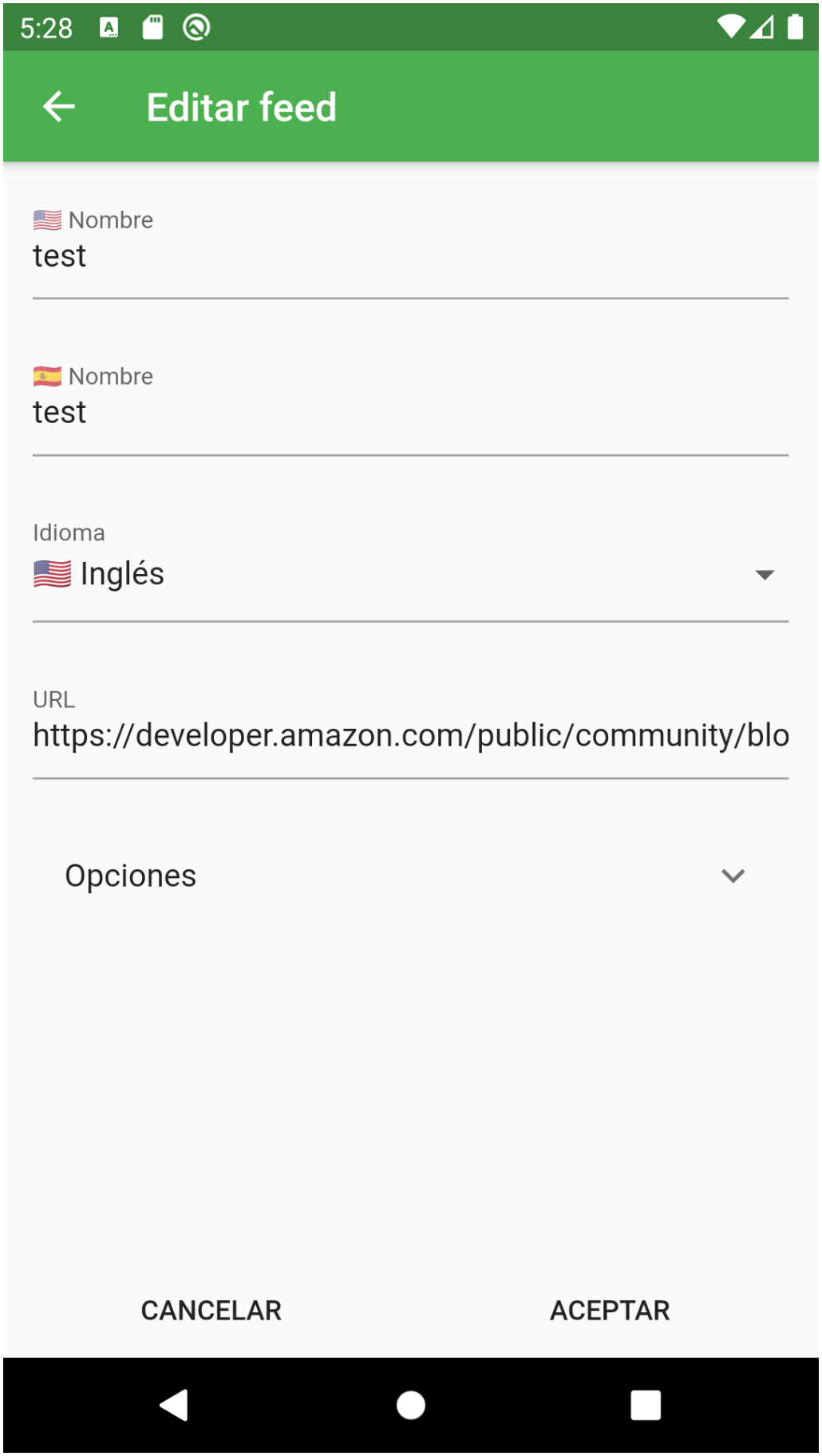


Figura B.3: App: Nueva/Editar feed

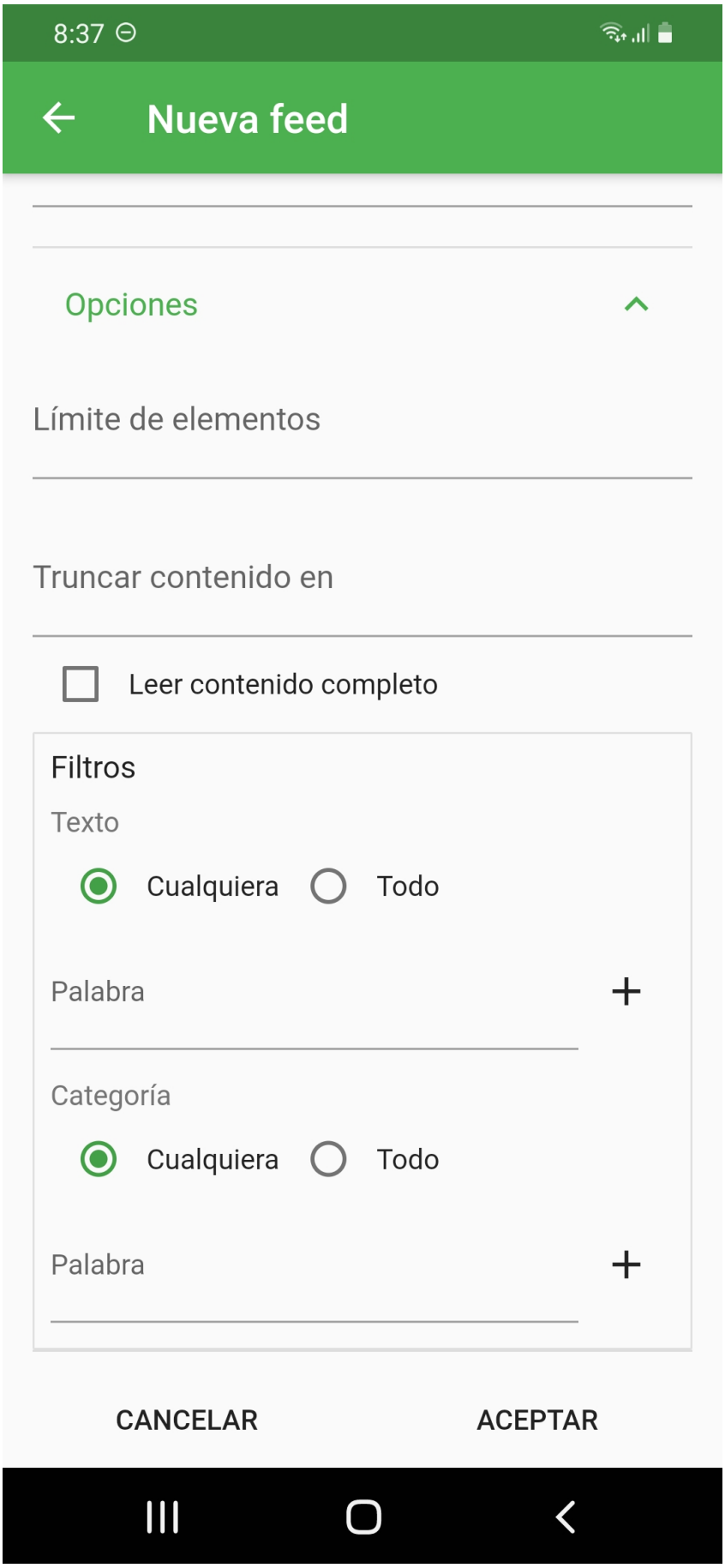


Figura B.4: App: Nueva/Editar feed - Opciones

APÉNDICE C
Glosario

Feed: Listado de artículos web con un formato estandarizado, normalmente utilizado en blogs o sitios de noticias para poder difundir sus contenidos fácilmente en otros servicios o páginas web. Estos suelen ser actualizados cada cierto tiempo.

ASK: (Acrónimo de "*Alexa Skills Kit*") Conjunto de herramientas proporcionadas por Amazon para el desarrollo de skills en Alexa.

CLI: (Acrónimo de "*Command Line Interface*"): Programa desarrollado para ser utilizado mediante la línea de comandos.